

Java 开发专家

# Java

## 面向对象编程

Helping readers to master OOP(Object-Oriented Programming)  
with Java and get deep understanding of how the language works

全书贯穿六条主线:

面向对象编程思想

Java语言的语法

Java虚拟机执行Java程序的原理

在实际项目中的运用

设计模式

性能优化技巧



孙卫琴

飞思科技产品研发中心 编著

监制



电子工业出版社

PUBLISHING HOUSE OF ELECTRONICS INDUSTRY  
<http://www.phei.com.cn>

## 内容提要

本书内容由浅入深，紧密结合实际，利用大量典型实例，详细讲解 Java 面向对象的编程思想、编程语法和设计模式，介绍常见 Java 类库的用法，总结优化 Java 编程的各种宝贵经验，深入阐述 Java 虚拟机执行 Java 程序的原理。全书内容包括面向对象的编程思想、Java 语言的基础知识、异常处理、类与对象的生命周期、多线程、Java 集合、输入/输出和 GUI 编程、图形图像处理、网络通信等。其最大特色是以六条主线贯穿全书：面向对象编程思想、Java 语言的语法、Java 虚拟机执行 Java 程序的原理、在实际项目中的运用、设计模式和性能优化技巧。同时，本书还贯穿了 Sun 公司的 SCJP (Sun Certified Java Programmer) 认证的考试要点。

书中实例源文件请到 <http://www.fecit.com.cn> 的“下载专区”下载。本书适用于所有 Java 编程人员，包括 Java 初学者及资深 Java 开发人员；亦可作为高校的 Java 教材，企业 Java 的培训教材，以及 Sun 公司的 SCJP 认证的辅导材料。

# 目录

第 1 章 Java 语言快速入门 .....	1
1.1 Java 简介 .....	1
1.1.1 Java 的起源与发展 .....	1
1.1.2 Java 语言特点 .....	1
1.1.3 Java 的用途 .....	4
1.1.4 用于 Web 的 Applet .....	4
1.1.5 独立运行的 Application .....	5
1.2 Java 程序开发 .....	5
1.2.1 Java 程序开发步骤 .....	6
1.2.2 Java 编译器 .....	6
1.2.3 Java 解释器 .....	7
1.2.4 Applet 查看器 .....	7
1.3 Java Applet 应用 .....	8
1.3.1 Java 和 Web .....	8
1.3.2 第一个 Java Applet .....	9
1.3.3 将 Applet 嵌入 HTML .....	10
1.3.4 一个实用 Applet .....	11
1.4 Java Application 应用 .....	13
1.4.1 字符方式的 Application .....	13
1.4.2 图形方式的 Application .....	14
1.4.3 Java 编程小结 .....	16
1.5 面向对象编程初步 .....	17
1.5.1 对象 .....	17
1.5.2 消息 .....	18
1.5.3 类 .....	18
1.5.4 继承 .....	18
1.5.5 接口 .....	19

习题 .....	19
----------	----

## 第2章 Java 基本语法 .....

21
----

2.1 Java 语言的基本组成 .....	21
2.1.1 标识符 .....	21
2.1.2 关键字 .....	22
2.1.3 分隔符 .....	22
2.2 数据类型 .....	23
2.2.1 基本数据类型 .....	23
2.2.2 直接量 .....	24
2.2.3 变量 .....	25
2.3 运算符与表达式 .....	28
2.3.1 赋值运算符 .....	28
2.3.2 算术运算符 .....	29
2.3.3 关系运算符 .....	30
2.3.4 条件运算符 .....	30
2.3.5 逻辑运算符 .....	31
2.3.6 位运算符 .....	32
2.3.7 其他运算符 .....	34
2.3.8 运算符的优先级 .....	34
2.4 数组 .....	35
2.4.1 一维数组的声明 .....	35
2.4.2 一维数组的创建与赋值 .....	36
2.4.3 多维数组 .....	37
习题 .....	38

## 第3章 Java 语句及其控制结构 .....

41
----

3.1 Java 程序结构 .....	41
3.1.1 Java 程序构成 .....	41
3.1.2 Java 语句 .....	42
3.2 选择语句 .....	43
3.2.1 if 语句 .....	43
3.2.2 if...else 语句 .....	43



3.2.3	if...else 复合结构 .....	44
3.2.4	switch 开关语句 .....	46
3.3	循环语句 .....	48
3.3.1	for 循环语句 .....	48
3.3.2	while 循环语句 .....	49
3.3.3	do...while 循环语句 .....	50
3.3.4	循环语句的嵌套 .....	51
3.3.5	循环语句小结 .....	52
3.4	跳转语句 .....	52
3.4.1	break 语句 .....	53
3.4.2	带标号的 break 语句 .....	53
3.4.3	continue 语句 .....	54
3.4.4	带标号的 continue 语句 .....	55
3.4.5	return 语句 .....	57
习题	.....	58

## 第4章 面向对象编程 ..... 61

4.1	类 .....	61
4.1.1	类的创建 .....	61
4.1.2	类的修饰 .....	65
4.1.3	类体 .....	66
4.1.4	类的构造方法 .....	66
4.2	成员变量 .....	67
4.2.1	成员变量的声明 .....	67
4.2.2	成员变量的修饰 .....	69
4.3	成员方法 .....	73
4.3.1	成员方法的设计 .....	73
4.3.2	成员方法的声明与修饰 .....	74
4.3.3	方法体 .....	77
4.3.4	消息传递 .....	78
4.4	对象实例化 .....	80
4.4.1	创建对象 .....	80
4.4.2	使用对象 .....	81
4.4.3	清除对象 .....	82
习题	.....	82

## 第5章 类的继承性和多态性 ..... 85

5.1 类的继承 .....	85
5.1.1 父类和子类 .....	85
5.1.2 成员变量的继承和隐藏 .....	86
5.1.3 成员方法的覆盖 .....	88
5.1.4 this 和 super .....	89
5.2 类的多态 .....	92
5.2.1 成员方法的重载 .....	92
5.2.2 构造方法的重载 .....	94
5.3 进一步讨论的问题 .....	96
5.3.1 对象的克隆 .....	96
5.3.2 子类对象和父类对象的关系 .....	98
5.3.3 类的包容 .....	100
习题 .....	104

## 第6章 包、接口和异常 ..... 105

6.1 程序包 .....	105
6.1.1 声明自己的程序包 .....	105
6.1.2 程序包的引用 .....	106
6.1.3 Java 的系统程序包 .....	106
6.2 接口 .....	108
6.2.1 声明接口 .....	108
6.2.2 接口的继承关系 .....	109
6.2.3 在类中实现一个接口 .....	110
6.2.4 在类中实现多个接口 .....	110
6.3 异常处理 .....	111
6.3.1 什么是异常 .....	111
6.3.2 异常发生的原因 .....	111
6.3.3 编译时对异常情况的检查 .....	112
6.3.4 异常的层次结构 .....	112
6.3.5 Java 定义的标准异常类 .....	113
6.3.6 异常的处理 .....	114
6.3.7 创建自己的异常 .....	117
6.3.8 throw 语句 .....	118

6.3.9 throws 语句 .....	119
6.3.10 finally 语句 .....	121
习题 .....	122

## 第7章 常用系统类 ..... 123

7.1 Applet 类 .....	123
7.1.1 Applet 简介 .....	123
7.1.2 Applet 的生命周期 .....	124
7.1.3 HTML 和 Applet 的参数传递 .....	126
7.1.4 APPLET 标签属性 .....	128
7.1.5 Applet 与 Application 的合并运行 .....	129
7.2 字符串类 .....	130
7.2.1 字符串与字符串类 .....	131
7.2.2 字符串类的构造方法 .....	132
7.2.3 String 类的应用 .....	134
7.2.4 StringBuffer 类的应用 .....	137
7.3 标准输入/输出 .....	138
7.3.1 标准输入方法 .....	139
7.3.2 标准输出方法 .....	140
7.4 其他常用类 .....	142
7.4.1 数学函数类 Math .....	142
7.4.2 日期类 .....	142
7.4.3 随机数类 Random .....	144
7.4.4 向量类 Vector .....	146
习题 .....	148

## 第8章 图形用户界面 ..... 151

8.1 组件 .....	151
8.1.1 标签 .....	151
8.1.2 按钮 .....	152
8.1.3 选项框 .....	153
8.1.4 复选框和选项按钮 .....	154
8.1.5 列表框 .....	156
8.1.6 文本框 .....	158
8.1.7 文本区 .....	158

8.1.8	滚动条	160
8.2	组件布局管理	161
8.2.1	顺序布局	161
8.2.2	边界布局	161
8.2.3	卡片布局	162
8.2.4	网格布局	163
8.2.5	网格包布局	164
8.2.6	面板的使用	166
8.2.7	手工布局	168
8.3	事件处理	169
8.3.1	Java 的事件处理机制	169
8.3.2	事件处理实例	171
习题		180

## 第9章 窗口、菜单和对话框 ..... 183

9.1	窗口	183
9.1.1	创建可关闭窗口	183
9.1.2	关于事件裁剪器	185
9.1.3	在窗口中加入组件	186
9.1.4	多重窗口	188
9.2	菜单	189
9.2.1	为窗口加入菜单	189
9.2.2	菜单综合应用	191
9.2.3	弹出式菜单	193
9.3	对话框	194
9.3.1	自定义对话框	194
9.3.2	文件对话框	198
9.4	Swing 简介	201
9.4.1	Swing 按钮与标签	202
9.4.2	Swing 工具栏	203
习题		206

## 第10章 图形处理 ..... 207

10.1	基本图形	207
10.1.1	直线	207

10.1.2	矩形	208
10.1.3	椭圆	210
10.1.4	圆弧	211
10.1.5	多边形	211
10.2	画布	212
10.2.1	创建画布	213
10.2.2	在画布上手工画图	214
10.3	文字输出	217
10.3.1	字符串、字符和字节输出	217
10.3.2	字体控制	218
10.4	颜色与绘图模式控制	221
10.4.1	颜色控制	221
10.4.2	绘图模式控制	224
习题		226

## 第 11 章 多媒体编程 ..... 227

11.1	图像处理	227
11.1.1	图像种类	227
11.1.2	图像的显示	228
11.1.3	幻灯机效果	231
11.1.4	生成图像	232
11.1.5	图形旋转与透明处理	233
11.2	动画处理	235
11.2.1	动画原理	235
11.2.2	图形双缓冲	236
11.2.3	用线程实现动画	238
11.2.4	文字的动画显示	240
11.2.5	图像高级处理——水中倒影	241
11.3	数字音频	244
11.3.1	加载声音文件	244
11.3.2	在 Applet 中播放声音	245
习题		247

## 第 12 章 流、多线程和网络编程 ..... 249

12.1	流处理	249
------	-----	-----

12.1.1	流 .....	249
12.1.2	流的分类 .....	250
12.1.3	数据流的应用 .....	252
12.1.4	文件类 .....	258
12.2	多线程 .....	262
12.2.1	线程与多线程 .....	262
12.2.2	创建线程 .....	263
12.2.3	线程的生命周期 .....	265
12.2.4	线程的优先级 .....	267
12.2.5	线程同步 .....	268
12.2.6	多线程问题 .....	272
12.3	网络编程 .....	273
12.3.1	Java 网络基础知识 .....	273
12.3.2	URL 编程 .....	273
12.3.3	创建 URL 对象 .....	275
12.3.4	利用 URL 读取服务器文件 .....	276
12.3.5	利用 URLConnection 和服务端交互 .....	277
12.3.6	利用 Socket 和服务端交互 .....	279
12.3.7	利用 Datagram 和服务端交互 .....	284
	习题 .....	288



# 第1章

## Java 语言快速入门

Java 是非常具有吸引力的面向对象编程语言,又是当前最流行的网络编程语言。Java 的出现引起了软件开发的变革,为迅速发展的 IT 业增添了新的活力。本章将介绍 Java 语言的基本特点和应用开发,给你一个 Java 概貌。

### 1.1 Java 简介

Java 语言的出现是为了解决当今计算机编程实践中碰到的许多问题。由于 Java 独具特点,使它在短短 5 年时间里就成为十分流行的计算机编程语言。

#### 1.1.1 Java 的起源与发展

Java 是一种计算机程序语言,用来编写嵌入在 Web 网页中运行的 Java Applet,也可以编写独立运行的 Java Application,是当今十分流行的网络编程语言。

Java 是 Sun 公司于 20 世纪 90 年代初开发的,最初并不是为了用于 Internet,而是作为一种小家用电器的编程语言,用来解决诸如电视机、电话、闹钟、烤面包机等家用电器的控制和通讯问题,命名为 Oak。由于这些智能化家用电器的市场需求没有预期的高,Sun 放弃了该项计划。就在 Oak 几近夭折之时,Internet 异常火爆起来。Sun 看到了 Oak 在计算机网络上的广阔应用前景,他们改造了 Oak,于是 Java 诞生了。

1995 年 5 月 Sun 正式发布了 Java。由于 Internet 上存在着巨大的商业利益,Java 的出现引起了商界的极大兴趣。作为专为商业用途设计的程序语言,Java 伴随着 Internet 的迅猛发展而发展,逐渐成为重要的 Internet 编程语言。由于 Java 提供了强大的图形、图像、动画、音频、视频、多线程及网络交互能力,使它在设计交互式、多媒体网页和网络应用程序方面大显身手,成为当今推广最快的一门计算机程序语言。

#### 1.1.2 Java 语言特点

按照 Java 设计者的解释,Java 是一个简单、面向对象、网络适用、解释型、健壮、安全、结构中立、可移植、高性能、多线程、动态的计算机程序语言。

## 1. 简单性

设计 Java 语言的出发点就是容易编程,不需要深奥的知识。Java 语言的风格十分接近 C++ 语言,但要比 C++ 简单得多。Java 舍弃了一些不常用的、难以理解的、容易混淆的成分,如运算符重载、多继承等。增加了自动垃圾搜集功能,用于回收不再使用的内存区域。这不但使程序易于编写,而且大大减少了由于内存分配而引发的问题。

简单性还体现在小上。Java 解释器、系统模块和运行模块都比较小,适合在小型机器上运行,也适合从网上下载。

## 2. 面向对象

面向对象编程是一项有关对象设计和对象接口定义的技术,或者说是一项如何定义程序模块才能使它们“即插即用”的技术。举例来说,一个“面向对象”的木匠更关心的是他正在做的椅子,其次才是使用的工具;而一个“非面向对象”的木匠首先关心的是工具,然后才是椅子。

Java 继承了 C++ 面向对象技术的核心,更具有动态解决问题的特性。对象通过继承和重定义,成为解决新问题的模块,使代码重用有了可能。

## 3. 网络适用性

Java 提供了大量的系统模块支持基于 TCP/IP 协议的编程,这使得 Java 建立网络连接要比 C/C++ 容易得多。Java 程序通过 URL 访问网络资源和存取本地文件系统一样简单。

## 4. 健壮性

Java 程序的健壮性从多方面得到了保证。Java 提供早期的编译检查和后期的动态(运行期)检查,大量消除了引发异常的条件。

Java 和 C++ 的显著不同是有一个指针类,它可以防止内存覆盖和数据破坏。另一个不同是 Java 用真正的数组代替了 C++ 的指针运算,可以进行数组元素的越界检查。Java 程序在没有授权的情况下是不能访问内存的。所有这些措施,使 Java 程序员不用再担心内存的崩溃,因为根本就不存在这样的条件。

## 5. 安全性

Java 主要用于网络和分布式环境,采取了很多措施来加强系统的安全性。Java 可以组建病毒无法入侵和无法篡改的系统,其数字验证使用了基于公共密钥的技术。

安全性和健壮性密切相关。Java 的指针类技术杜绝了非法存取数据结构或关键对象属性的可能,关死了病毒发起攻击的大门。

## 6. 结构中立

网络一般由各种类型的计算机构成,Internet 尤为如此。为了使 Java 程序在网络的

任何地方都能运行,Java 编译器生成的目标代码是结构中立的,即任何安装了 Java 运行环境的计算机都能执行这种代码。这一点对单机系统也同样重要。很多软件都要针对不同的平台(如 IBM、苹果机等)开发不同的版本,而 Java 程序的同一个版本就可运行在任何平台上。

这种目标代码称为字节码(bytecode),它和计算机平台无关。相反,字节码被设计成既能很容易地被任何计算机解释执行,又能快速地翻译成本地机器代码。

## 7. 可移植性

结构中立构成了程序可移植性的基础。另一方面,很多语言的基本数据类型长度都有平台依赖性,而 Java 则采取固定长度。例如整数类型 Int 的长度固定为 32 位,双精度类型 Double 的长度固定为 64 位。

Java 的类库提供了可移植的接口。例如,类库中有一个抽象类 Window,它适用于 UNIX、Windows NT/95 和 Macintosh。Java 系统本身也是可移植的。Java 编译器是用 Java 写成的,Java 解释器是用 ANSI C 写成的,它们都有良好的移植性。

## 8. 解释型

Java 是解释执行的。程序运行时,字节码被直接翻译成本地机器指令,中间没有存储。由于模块连接是步进的和多线程的,执行速度可以很快。

## 9. 高性能

设计字节码时已经把机器码的翻译问题考虑进去了,所以实际翻译过程非常简单,编译器在对程序进行优化后生成高性能的字节码。

尽管字节码翻译执行的速度已经足够快,但有时也会要求有更高的性能。程序运行时,字节码将被快速翻译成当前 CPU 的指令,在某种程度上相当于将最终机器指令的产生放在动态加载器中进行。在 Sun Microsystems SPARCStation 10 计算机上进行的一项 30 万个方法调用的实验,证明解释型字节码翻译成机器代码的速度和 C/C++ 几乎没有区别。

## 10. 多线程

现实世界中,每时每刻都有很多事情在我们身边同时发生。多线程的概念和这种情况差不多,就是让计算机同时运行多个程序段。编写一个能同时处理多个任务的程序要比编写一个单线程程序困难得多。

Java 提供了一套复杂的线程同步化机制,程序员可以方便的使用基于这种机制设计的方法,编写出健壮的多线程程序。

## 11. 动态性

在很多方面,Java 都比 C/C++ 更加动态化,它被设计成能适应环境变化的语言。例如,C++ 属于编译加载,不同版本的 C++ 或类库的升级通常会使得软件开发商重

新编译他们的程序。对最终用户而言,问题可能会更严重。而 Java 属于运行加载,Java 的类库(即插即用模块集)可以自由添加方法和属性而不会影响到用户程序。因此,Java 的动态性可以更好地适应不断变化的执行环境。

### 1.1.3 Java 的用途

Java 程序有两种类型:

一种是可在 Web 网页上运行的 Applet,称为小应用程序。考虑到网络环境、连接速度等原因,Applet 一般都比较小,适合客户端下载。很多网站利用 Java 开发出了商业网络平台,实现交互运行。还有大量的 Applet 嵌入到网页,使页面变得更加活泼生动。Applet 不能单独运行,必须嵌入在 HTML 文件中,由 Web 浏览器执行。

另一种是 Application,即应用程序,可完成任何计算任务。运行时不必借助于 Web 浏览器,可单独执行。

目前,Sun 正在把 Java 的目标从传统的计算机应用向其他数字设备领域扩展。这似乎是一种回归,Java 又回到了它的起点。例如,Java 平台技术 J2ME 可用于发展中的无线网络系统,支持可上网手机浏览 Internet,收发电子邮件。Sun 推出的基于 Java 的 JINI 技术,能使各种小型数字设备以非常简单的方式连接到任意“无准备”网络,访问新网络变得像接入电话机一样简单。

可以预见,在不远的将来,我们将会使用更多的“Java 设备”,从数字手机、电视机顶盒到传统的家用电器,还有更多叫不出名字的创新产品。

### 1.1.4 用于 Web 的 Applet

图 1.1 是 Java JDK 自带的一个演示程序。用浏览器打开网页文件 example1.html,

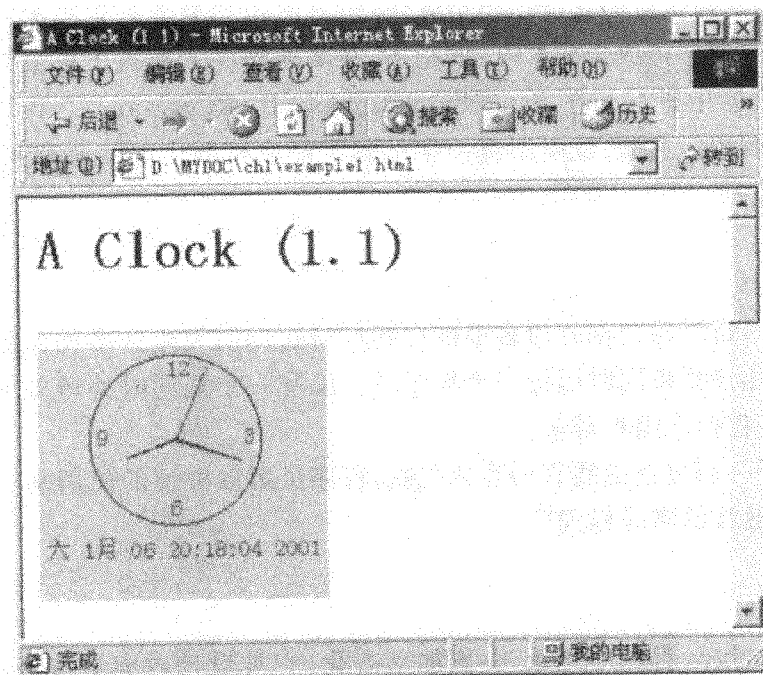


图 1.1

嵌入的 Applet 就开始运行,一个模拟时钟出现在网页上,显示出系统当前时间。

读者可能要问,Applet 有什么用途呢? Applet 可以嵌入网页,实现 HTML 不具备的一些功能,你可以把 Applet 看成是 Web 的好朋友。

Internet 上的 Web 页面是用 HTML(超文本标记语言)编写的,HTML 只能用来定义页面的布局结构,它不是一种编程语言。用 HTML 建立的 Web 页面是静态的,而且不具备交互能力。假如想通过 Web 页从事商业活动,就必须使 Web 页具有和用户交互的能力,此时 Applet 就可以大显身手了。例如,用 Java 编写一个接受用户订单输入的 Applet,然后嵌入到 HTML 中,当用户打开该页面时,嵌入的 Applet 将被运行,用户可以输入订单信息,然后安全地发送出去。

传统的方法是通过在网页中添加嵌入式编程语言脚本(如 CGI,JavaScript)实现交互能力,但这些嵌入式编程语言的能力有限,而且代码是公开的,远不如 Java 的强大功能和保密性能。Applet 尽管是嵌入到 HTML 中的小应用程序,但 Java 语言的全部功能都可以实现,能解决一些传统编程语言很难解决的问题,例如多线程、网络连接、分布式计算等等。

### 1.1.5 独立运行的 Application

Applet 运行时的窗口界面是由浏览器提供的,因此它不能脱离浏览器而独立运行。而 Application 则和任何 Windows 应用程序一样自建窗口界面,可以独立运行。事实上,Java 语言就是一门高级编程语言,和其他高级编程语言并无两样。

图 1.2 所示为一个图形方式的 Java Application,运行结果和普通的 Windows 应用程序完全一样。

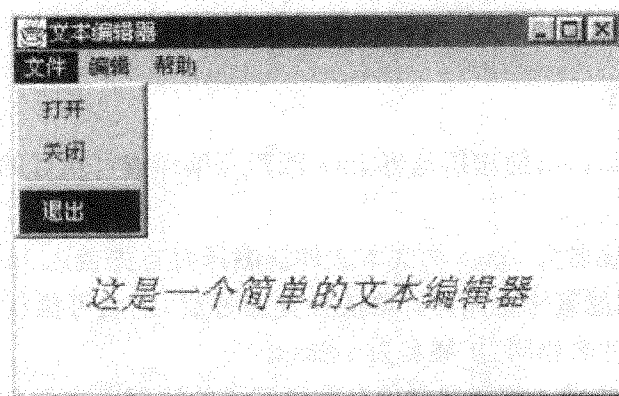


图 1.2

## 1.2 Java 程序开发

Sun 的 JDK 包含了一整套开发工具,对编程最有用的是 Java 编译器、Applet 查看器和 Java 解释器。不过这些工具都是命令行工具,你只能从命令行即 MS-DOS 提示符下运行它们。很多人可能会不习惯,但这是 Sun 特意采取的策略,为的是把精力更多地投

人到 Java 语言本身而不是花在开发工具上。

### 1.2.1 Java 程序开发步骤

要编写和运行第一个 Java 程序,需要有文本编辑器和 Java 开发平台。你可以使用操作系统提供的 Edit 或记事本作为编辑器,用 Java 2 作为开发平台。

2000 年 5 月, Sun 推出了最新版本 Java 2 开发工具包 JDK 1.3, 可以免费下载安装, 下载地址为: <http://java.sun.com/products/jdk>。本书介绍的内容均基于 Java 2, 给出的例子全部在 JDK 1.3 环境下通过验证。

开发一个 Java 程序有 3 个步骤:

(1) 建立 Java 源程序 Java 源程序包含 Java 命令语句, 可用任何文本编辑器建立。注意: 使用一些带格式的文本编辑器如 Word 等, 在保存源程序文件时, 应选择以 MS-DOS 文本格式保存。

(2) 编译源程序 在命令行状态下执行 `javac`, 将源程序编译成字节码文件, 字节码文件的内容是 Java 虚拟机(JVM)可执行的指令。编译时如果出现错误, 则终止编译, 直到修改程序错误最终通过编译为止。

(3) 运行 Java 程序 Java 虚拟机由 Java 解释器实现。在命令行状态下执行 Java, 可将 Application 字节码文件解释为本地计算机能够执行的指令并予以执行。

如果程序是 Java Applet, 应建立一个 HTML 文件, 在适当位置加入 Applet 字节码文件名, 并用 Applet 查看器或直接用浏览器打开 HTML 文件。Applet 的运行结果会在查看器或浏览器窗口中显示出来。

如果程序是字符方式的 Application, 运行结果在 MS-DOS 窗口中显示。如果是图形方式的 Application, 将自动返回 Windows 以显示图形界面。

### 1.2.2 Java 编译器

Java 编译器(`javac.exe`)的作用是将 Java 源程序编译成可执行的程序代码, 是最基本的开发工具。

Java 源程序是扩展名为 `.java` 的文本文件。编译时首先读入 Java 源程序, 然后进行语法检查, 如果出现问题就终止编译。语法检查通过后, 生成可执行程序代码即字节码, 字节码文件名和源文件名相同, 扩展名为 `.class`。

在 MS-DOS 窗口中键入编译器文件名和要编译的源程序文件名, 按回车键即开始编译。如果源程序没有错误, 则屏幕上没有输出, 否则, 将显示出错信息。

我们尝试一下如何编译 JDK 提供的演示程序 `Clock2.java`, 首先把演示程序拷贝到新建的子目录, 然后执行编译命令, 如图 1.3 所示。

注意: 在命令行下执行 DOS 命令, 最好是在 `AUTOEXEC.BAT` 文件中预设命令搜索路径。假设 JDK 安装在 `C:\JDK` 目录下, 则 Java 所有的命令文件均放在 `C:\JDK\BIN` 子目录下。可在 `AUTOEXEC.BAT` 文件中加入以下语句: `SET PATH = %PATH%; C:\JDK\BIN`。也可在 MS-DOS 窗口中选择“属性”按钮, 在程序标签下的“工作目录”中写入文件的所在路径, 如 `C:\JDK\BIN`。



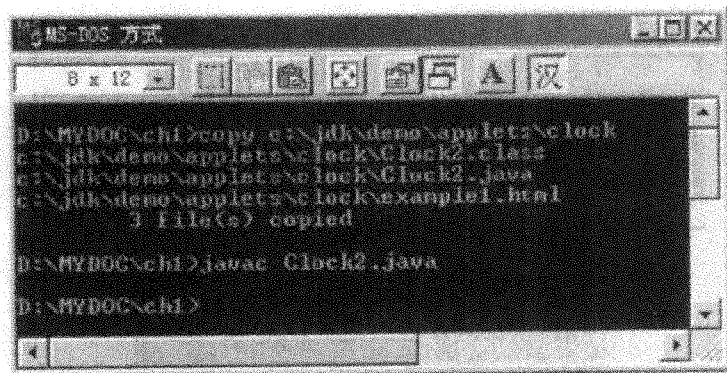


图 1.3

### 1.2.3 Java 解释器

Java 解释器(java. exe)负责将编译后的字节码解释为本地计算机代码。Java 的平台无关性不在于源程序中有什么奥妙,而在于每一种计算机上都安装了一个合适的解释器。解释器建立了一个运行平台即 Java 虚拟机,将不同计算机上的系统差别隐藏起来,使字节码面对一个相同的运行环境,实现了“写一次,到处用”的目标。

对于 Applet 来说,Web 浏览器或 Applet 查看器都有内置的解释器,并负责提供 Applet 所需要的图形界面。

Application 可通过解释器来运行。称其为独立,并不是说它可以像编译后的 VB 或 C++ 应用程序一样直接运行,而是指它可以脱离浏览器单独运行,不再需要 HTML 文件。运行 Application 十分简单,将应用程序编译成字节码后就可用 java. exe 运行。

对于图 1.2 中的窗口应用程序,运行命令如图 1.4 所示。注意:命令中的 WindowApp 是字节码文件名,不加扩展名,还要注意大小写,windowapp 不等于 WindowApp。



图 1.4

### 1.2.4 Applet 查看器

Applet 查看器(appletviewer. exe)实际上是一个模拟浏览器,可显示 Applet 的运行结果。使用 Applet 查看器比较方便,因为它仅显示有关 Applet 的内容,而浏览器通常还要显示 Web 页面的内容。对前述的 Clock 演示程序,在 MS-DOS 窗口作者所设置的 D:\MYDOC\ch1 路径下,键入命令:appletviewer example1. html,打开已有的 HTML 文件 example1. html,就可查看其在 Web 页中的模拟运行结果,如图 1.5 和图 1.6 所示。

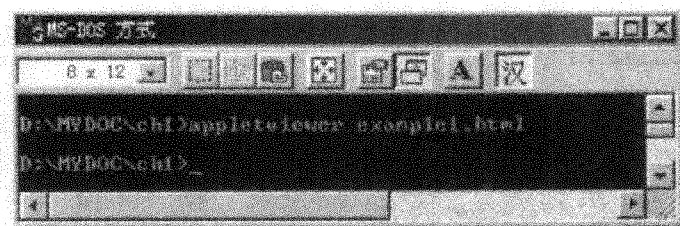


图 1.5

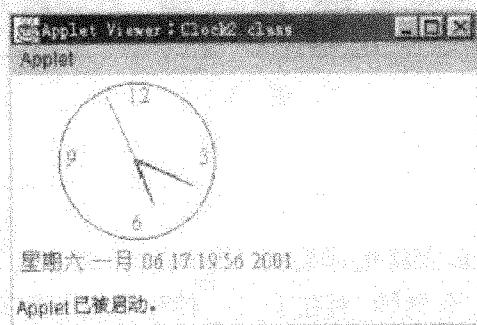


图 1.6

注意: Applet 查看器不能直接运行字节码文件, 而是打开含有 Applet 字节码的 HTML 文件。这个 HTML 文件需要另外建立, 它本身也是文本文件, 可用文本编辑器建立。

## 1.3 Java Applet 应用

当今的 Internet 上存在着数以十亿计的各类网页, Java 作为实现动态的、交互式网页功能的编程工具, 正在扮演着重要角色。Applet 不仅能为网页添加声音和动画效果, 还能实现客户机/服务器连接。

### 1.3.1 Java 和 Web

理解 Applet 和 HTML 的关系对应用 Applet 非常重要。HTML 是网页设计语言, 它采用一整套标记来定义 Web 页。一个 HTML 文件可定义一个 Web 页, 文件的扩展名为 .html 或 .htm。我们可用文本编辑器打开 HTML 源文件, 看一下 Web 页是如何设计的。

下面是一个含有 Java Applet 的 Web 页, 图 1.7 是运行中的 Web 页, 其中图像是动画显示, 图像上的文字是滚动显示。图 1.8 是源文件部分内容。

从图 1.8 中我们可以看到, Applet 的字节码文件名 panj.class 作为一个外部引用, 出现在标记: `<applet code="panj.class" WIDTH="302" HEIGHT="272">` 内。

在 HTML 文件中, 处理 Applet 和处理图像完全一样, 都是把它们作为外部引用。这是因为二者都是二进制的, 不能直接包含在纯文本的 HTML 文件中。当这个 Web 页被浏览器下载到本地计算机后, 其中的图像引用由浏览器负责显示, 而 Applet 则由浏览器

内置的 Java 解释器执行。

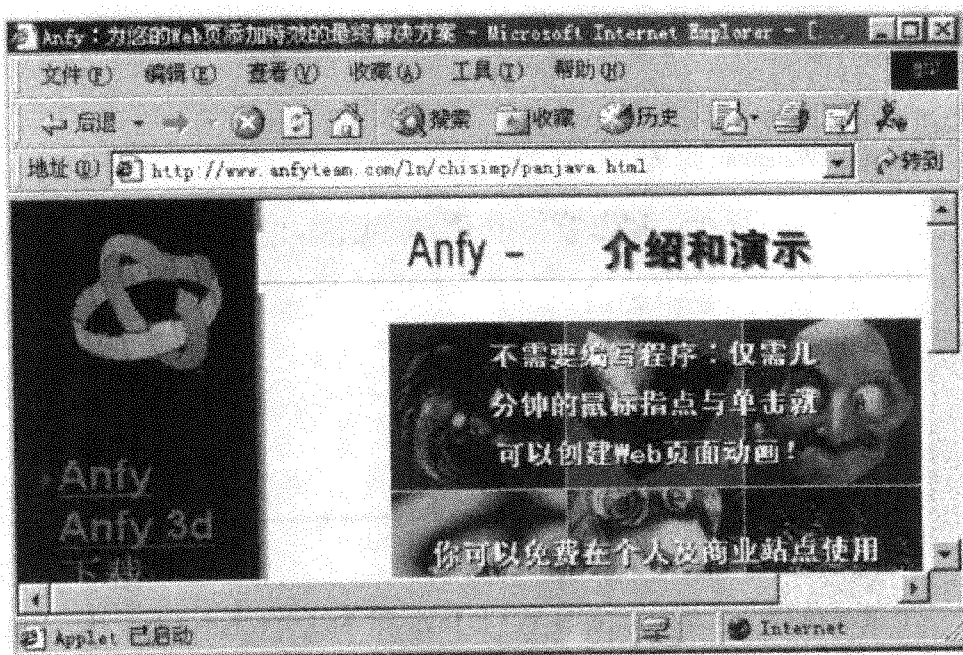


图 1.7

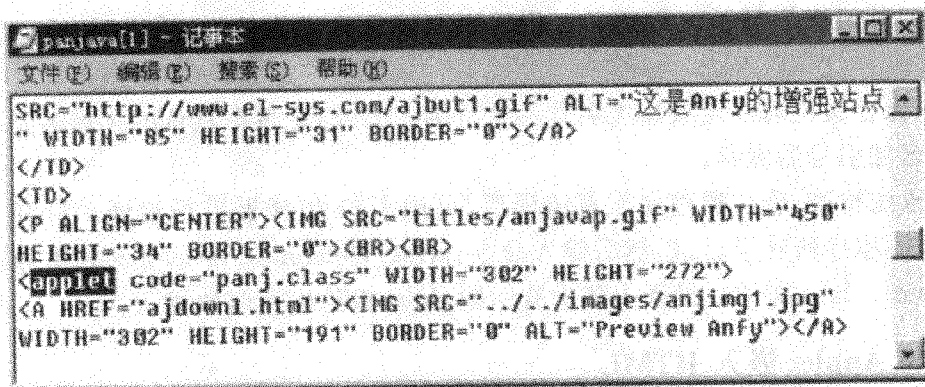


图 1.8

从本例中,我们可以看到 Applet 和 HTML 的关系。HTML 可以没有 Applet,但很多重要功能将无法实现,而 Applet 有赖于 HTML 才能运行,二者共同为 Internet 这个虚拟世界增添光彩。

### 1.3.2 第一个 Java Applet

**例 1.1** 第一个 Java Applet: Welcomel.java, 显示字符串“欢迎使用 Java 2”。运行结果如图 1.9。

```
/* My first Applet program
   in Java
*/
import java.applet.Applet; // 引入 Java 系统类 Applet
```

```

import java.awt.Graphics; // 引入 Java 系统类 Graphics
public class Welcome1 extends Applet {
    public void paint(Graphics g) {
        g.drawString("欢迎使用 Java 2", 20, 20);
    } // 结束 paint 方法定义
} // 结束类 Welcome1 定义

```

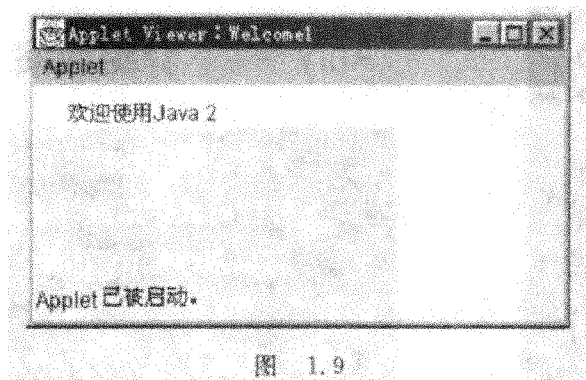


图 1.9

这是一个最简单的 Applet。我们注意到,Java 程序以类(class)为主体,这里定义了一个类 Welcome1。类可以继承另一个类,以便使用它的特性。类中定义了一个方法 paint,用来输出字符串“欢迎使用 Java 2”。头部是引入语句,用来引入 Java 系统类。程序中还是以两种格式添加了注释语句。用记事本建立这个文件,然后以“Welcome1.java”保存文件。

注意:保存文件时一定要用类名作为文件名。另外,记事本默认的扩展名是 txt,所以要给文件名加引号后保存。

经过语法检查,确认源程序没有错误后,可用 Java 编译器进行编译。打开 MS-DOS 窗口,进入源程序所在子目录,然后键入命令:javac Welcome1.java,如果没有错误即可得到 Welcome1.class 文件。编译中如果有问题,Java 将给出相应的提示。

### 1.3.3 将 Applet 嵌入 HTML

Applet 的运行依赖于浏览器的窗口界面,必须将 Applet 字节码文件嵌入到 HTML 文件中才能正常运行。下面,我们用记事本创建 HTML 文件,文件内容如图 1.10 所示。文件建好后,以“demo.html”为名保存到字节码文件所在目录。

HTML 语言不是本书要讨论的内容,运行一个 Applet,只需要使用 HTML 符号集

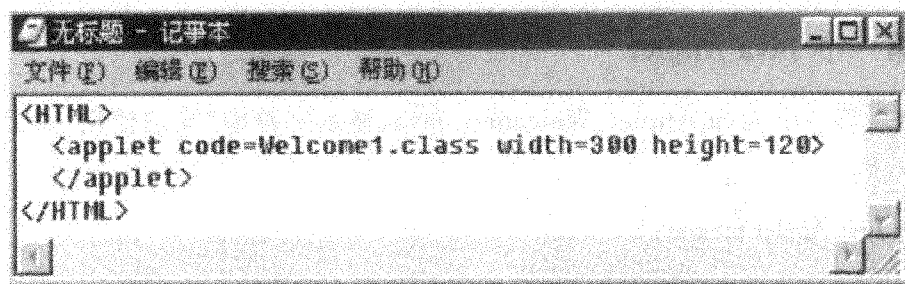


图 1.10

中的最小集合即可。HTML 标记包含在尖括号内,并且总是成对出现,前边加斜杠表明标记结束。我们用<HTML>和</HTML>来标记 HTML 文件的开始和结束,用<applet>和</applet>标记 Applet 的开始和结束。<applet>至少需要包含以下三个参数:

- code 指定要打开的 Applet 字节码文件名;
- width 指定 Applet 占用浏览器页面的宽度,以像素点为单位;
- height 指定 Applet 占用浏览器页面的高度,以像素点为单位。

注意:不是把字节码直接嵌入到 HTML 文件中,而是把字节码文件名嵌入到 HTML 文件中。字节码文件名可包含路径,否则,字节码文件应和 HTML 文件处于同一目录下。另外,HTML 对字符大小写是不敏感的,参数值可加引号也可不加。

### 1.3.4 一个实用 Applet

例 1.2 设计一个 Applet,能进行简单加法运算。运行结果如图 1.11。

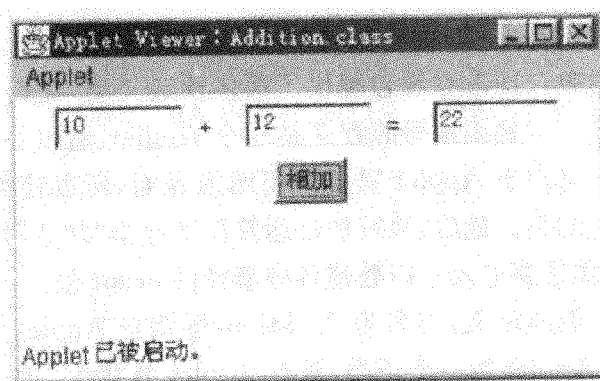


图 1.11

```
// 源程序: Addition.java  
import java.awt.*;  
import java.awt.event.*;  
import java.applet.Applet;
```

```
public class Addition extends Applet implements ActionListener {  
    Label label1=new Label("+");  
    Label label2=new Label("=");  
    TextField field1=new TextField(6);  
    TextField field2=new TextField(6);  
    TextField field2=new TextField(6);  
    Button button1=new Button("相加");
```

```
    public void init() { // 初始化  
        add(field1);  
        add(label1);  
        add(field2);
```

```

        add(label2);
        add(field3);
        add(button1);
        button1.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) { // 处理按钮事件
        int x=Integer.parseInt(field1.getText())+Integer.parseInt(field2.getText());
        field3.setText(Integer.toString(x)); // 数值转换为字符串
    }
}

```

我们来看程序的结构。前面 3 行是加载语句,分别引入 Java 系统包 awt 和 applet。语句 import java.applet.Applet 引入了 applet 中的 Applet,语句 import java.awt.\* 引入了 awt 的所有类,语句 import java.awt.event.\* 引入了 awt 的 event 包的所有类。

类可以理解为对象的抽象概念,是本书要深入讨论的概念之一。类就像一个模板,可用它生成一个实在的对象。Java 系统包提供了很多预定义类,我们可以直接引用它们而不必从头开始编写程序。本例要编写的程序是一个 Applet,所以一定要引入 Applet 类,以便使用其各种特性。又由于 Applet 需要使用图形界面,所以还需要加载 awt,其中包含了所有处理图形界面的类。最后,执行加法运算需要点击“加法”按钮,这会产生一个鼠标事件或键盘事件,因此还要引入专门处理各种事件的 event 包。

程序的主体是一个类的定义,类名为 Addition,继承自 Applet 类。继承不是目的而是一种手段,说明它是 Applet 的一个子类,具有 Applet 的共性。继承的属性和方法往往不足以实现程序的要求,必须根据题目的具体要求,由用户添加各种对象和方法,改造成满足题目要求的程序。Java 编程就是基于这样的思想。

语句 public class Addition extends Applet implements ActionListener 说明 Addition 是继承自 Applet 的公共子类,具有事件监听接口,在程序运行时可监听发生了什么事情,并负责调用相应的事件处理方法作出响应。这个框架出来后,剩下的任务就是装配程序。添加两个标签对象用于显示运算符号;添加三个文本域对象用于接受用户的输入;再添加一个按钮对象用于执行加法运算。

所有添加进来的对象行为都由这个类来控制,这种控制是通过改造继承下来的方法实现的。类所包含的方法相当于传统编程语言的过程或函数,可完成一定的功能。init 是一个不需要返回值的方法,它将各个对象加入到 Addition 的显示区,并把事件监听者(Addition)注册给按钮对象。事件监听者和按钮对象之间通过监听接口建立起信息通道,一旦按钮被点击,该事件的信息就会传递给监听者。监听者收到事件信息后,负责调用事件处理方法 actionPerformed 来响应。本例中的按钮事件处理方法就是将用户输入的两个数转换成整数后相加,然后显示在第三个文本域。

在此,我们仅从面向对象编程的角度,对本例的程序设计作出说明,程序语句中的具体含义有待于读者在后续章节进一步学习和掌握。



## 1.4 Java Application 应用

Application 有两种编程方式,一种是字符方式,另一种是图形方式。前者如同早期的 BASIC 或 C 语言程序,在 MS-DOS 环境下运行,后者则采用标准的图形界面运行,它们都不需要浏览器的支持。

### 1.4.1 字符方式的 Application

例 1.3 第一个 Java Application: Welcome2.java。运行结果如图 1.12。

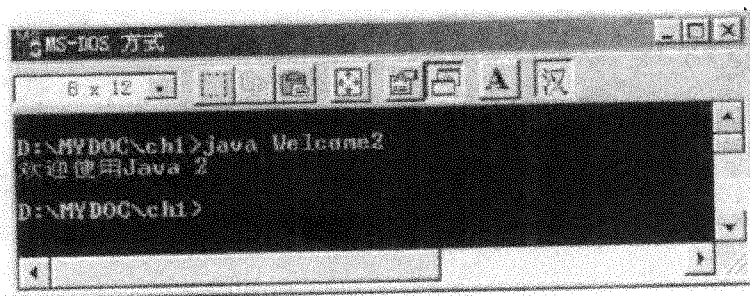


图 1.12

```
// My first Application program in Java
class Welcome2 {
    public static void main(String args[]) {
        System.out.println("欢迎使用Java 2");
    } // 结束 main 方法定义
} // 结束类 Welcome2 定义
```

我们看到 Application 也是以类的形式声明的,是一个自定义类。main 方法是程序运行的入口,并在该方法中调用系统标准输出方法 println 输出字符串“欢迎使用 Java 2”。

用记事本建立这个文件,在文件对话框中输入“Welcome2.java”保存文件。然后打开 MS-DOS 窗口,进入源程序所在子目录,键入命令: javac Welcome2.java 进行编译。如果没有错误即可生成 Welcome2.class 文件,键入命令: java Welcome2 可得到运行结果。

例 1.4 设计一个 Application,输入一个字符,然后把字符的 ASCII 码值显示出来。运行结果如图 1.13。

```
// 源程序: ValueOfChar.java
import java.io.*;

public class ValueOfChar {
    public static void main(String Args[]) {
        char ch='';
        System.out.print("请输入一个字符:");
```

```

try { // 异常处理块
    ch=(char)System.in.read();
}
catch (IOException e) {}
System.out.println("字符"+ch+"的 ASCII 值为:"+ (int)ch);
}
}

```



图 1.13

main 方法中定义了一个 char 型的局部变量 ch,用它来接收输入的字符,并直接使用 java.io 包中的系统标准输入/输出方法实现输入/输出。由于等待用户输入时会使程序流程中断而产生一个异常,程序中引入了异常处理机制 try...catch。(char)和(int)的作用是强制类型转换。Java 属于强类型语言,必须预先指定变量类型,但在使用中可进行强制类型转换,例如把 char 型的 ch 当作 int 型使用。

注意:Java 的字符串操作符“+”可将其他类型的数据转换成字符串。

#### 1.4.2 图形方式的 Application

字符方式的应用程序在 GUI(图形用户界面)占统治地位的今天已逐渐丧失用途,仅在一些特殊场合还能看到它的身影,绝大多数程序员已转向 GUI 编程。Java 提供了很强的图形界面处理能力,利用这些特性,程序员可编写出完善的图形界面 Application。图形方式的 Application 编程,使用了较多的 Java 语言要素,要比字符方式编程困难一些。

**例 1.5** 设计一个图形方式的 Java Application,每次产生 10 个介于 0 和 100 之间的随机数,用冒泡排序法排序。运行结果如图 1.14。

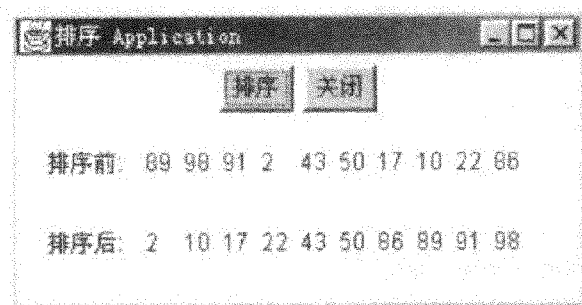


图 1.14

```

// 源程序: BubbleSort.java
import java.awt. * ;
import java.awt.event. * ;

public class BubbleSort {
    public static void main(String args[]) {
        new Frame1();
    }
}

class Frame1 extends Frame implements ActionListener {
    int arr[]=new int[10];
    Button button1=new Button("排序");
    Button button2=new Button("关闭");

    Frame1() {                // 构造方法
        super("排序 Application");
        setLayout(new FlowLayout());
        add(button1);
        add(button2);
        button1.addActionListener(this);
        button2.addActionListener(this);
        setSize(300,150);
        show();
    }

    public void paint(Graphics g) {    // 继承的画出方法
        g.drawString("排序前:",20,60);
        g.drawString("排序后:",20,100);
        for (int i=0;i<10;i++) {
            arr[i]=(int)(Math.random()*100);
            g.drawString(Integer.toString(arr[i]), 70+i*20, 60);
        }
        sort();
        for (int i=0;i<10;i++) {
            g.drawString(Integer.toString(arr[i]), 70+i*20, 100);
        }
    }

    public void sort() {              // 自定义的排序方法
        int temp;
        for (int j=1;j<arr.length;j++)
            for (int i=0;i<arr.length-1;i++)
                if (arr[i]>arr[i+1]) {

```

```

        temp=arr[i];
        arr[i]=arr[i+1];
        arr[i+1]=temp;
    }
}

public void actionPerformed(ActionEvent e) { // 继承的事件处理方法
    if (e.getSource() == button1)
        repaint();
    else
        System.exit(0);
}
}

```

程序中定义了两个类,含有 main 方法的 BubbleSort 是主类,只做一件事情,将子类 Frame1 实例化,起到了打开程序运行入口的作用。Frame1 继承自 Frame 类并建立了事件监听接口。Frame 类是图形界面最重要的类,它可在图形界面操作系统上创建标准窗口。唯一的不能自动关闭,它不能响应用户单击窗口右上方的关闭按钮,需要在程序中专门编写关闭方法。Frame1 添加了三个对象——数组 arr 和两个按钮。定义了构造方法、画出方法和事件响应方法,另外添加了一个自定义方法进行排序。

数组 arr 作为 Frame1 的对象可由类的方法引用,分别由 paint 方法和 sort 方法调用,实现排序和输出。button1 和 button2 可产生鼠标点击事件,Frame1 监听到事件后,判断哪一个按钮是事件源,然后执行相应的动作。点击“排序”按钮时,将执行 repaint 方法,它将再次调用 paint 方法重新生成一组随机数,排序后画出来;如果点击“关闭”按钮,将释放所有对象并返回操作系统。

在构造方法中,用 super 设定了主窗口的标题,用 setLayout 指定构件的布局方式。Java 对窗口中的组件采取不固定位置的方式,对于 Frame 类来说,组件布局默认设置为沿窗口四边摆放,本例中改为浮动布局,不管窗口大小如何改变,始终保持按钮位于窗口中间。

需要注意的是,Frame 对实例化对象没有指定默认大小,必须在初始化时设定大小和可见,否则窗口将不可见。

图形界面中,对象的显示由 paint 方法实现。覆盖这个方法,添加适当的代码,就可实现输出要求。paint 中使用了 Graphics 类的字符串输出方法,它具有定位输出功能。本例的 paint 方法采用了两个循环结构,第一个循环结构生成 10 个随机数,并逐个显示出来。然后调用排序方法对这 10 个随机数排序,排序结果由第二个循环结构显示出来。使用的随机数生成方法和整数转字符串方法来自于 java. lang,程序运行时可自动引入这个包,因此程序中没有这个包的引入语句。

### 1.4.3 Java 编程小结

以下是 Java 编程要点:

1.16 •

- 源程序的文件名和程序中定义的主类名应保持一致,包括字母大小写的匹配;
- Java 严格区分大小写,例如 applet 和 Applet 代表了不同的含义;
- 语句以分号结束;
- 程序中可加注释,用双斜杠“//”引导,“/\* \* /”可包含多行注释;
- 语句体(类体、方法体、结构体等)以大括号界定。
- 保持良好的书写风格,不同级别的语句最好采取缩进的方法来表示它们的差异。

如果读者熟悉 C++,就会发现 Java 源程序和 C++ 很相似。事实上,它们的风格和结构确实是一致的,这意味着从 C++ 转向 Java 或从 Java 转向 C++ 编程是毫不困难的。

## 1.5 面向对象编程初步

从上面的例子中可以体会到,Java 是面向对象编程语言。面向对象编程(OOP)是一种全新的编程理念,如果从来没有过 OOP 编程经验,你需要从基本概念开始。什么是对象?类又是什么?它们之间有什么关系?对象是如何通过消息和外部通讯的?下面,我们就讨论这些问题。

### 1.5.1 对象

对象(object)是相关数据和方法的结合体。软件对象是现实世界对象的抽象模型。

现实世界中,对象随处可见。一辆自行车,一只可爱的小狗,一台计算机,它们都属于对象。对象普遍具有的特征是:状态和行为。小狗有状态(名字、颜色、品种、饥饿)和行为(叫唤、撒欢儿、吃食),自行车有状态(变速器、脚蹬、车轮、刹车器)和行为(加速、刹车、换挡)。从现实世界中抽象出来的软件对象也具有同样的状态和行为特征。软件对象通过变量维持其状态,通过方法实现其行为。变量是一种有名称的数据实体,而方法则是和对象相关的函数或过程。

软件对象可以表达现实世界中的对象,例如在动画程序里用一个小狗模型代表现实世界里的小狗。软件对象还可以表达现实世界中的一些抽象概念,例如图形用户界面的事件就是一个常用的对象,代表了用户点击鼠标或按下键盘的动作。如果给定了动画程序里小狗的名字、形状和移动速度、移动方法,我们就有了一个确定的对象,称为实例。相应地,和一个实例对象相关的变量称为实例变量,相关的方法称为实例方法。

在软件对象中,变量处于核心,方法包围着变量,将它们隐藏起来使外界不可见。这种封装正是 OOP 设计者追求的理想境界。然而,事情并不总是这样,有时也需要对象公开一部分变量和方法。在 Java 中,通过对象的访问控制达到这一目的。相关数据和方法封装到一个包里,为程序员带来了两个好处:模块化和数据隐藏。模块化意味着对象源代码的编写和维护可以独立进行,不会影响到其他模块,而且有很好的重用性。数据隐藏则使对象有能力保护自己,自行维护自身的数据和方法而不影响所有依赖于它的对象。对象提供一个公共接口和其他对象联系。

### 1.5.2 消息

单独一个对象是没有什么作用的,多个对象联系在一起才会有完整的功能。那么对象是靠什么驱动的呢?对象之间的相互联系和相互作用是靠消息(message)的传递。

在例 1.2 中我们看到,点击加法按钮,一个事件消息就传递给事件监听者。事件监听者调用事件处理方法进行响应,最后将处理结果以消息的形式传递给文本域对象。文本域用自身的方法接收消息,并将结果显示出来。由此看出,对象的行为由方法来实现,消息传递是对象之间进行交互的主要方式。构成消息的三个要素是:接收消息的对象;接收消息的方法;方法所需要的参数。

### 1.5.3 类

现实世界中有很多同类对象。例如你的自行车是千千万万辆自行车中的一个,用 OOP 术语说,是自行车类的一个实例。自行车类有很多共同特征(状态和行为),但你的自行车和其他自行车是有区别的。制造自行车的厂商不会为每一辆自行车都设计一张图纸,然后按照这张图纸造出这辆自行车。他们通常是设计一份图纸,然后造出一批自行车,这是工业效率的基本要求。

基于同样道理,OOP 总结对象的特征设计成类(class)。类就是对象的软件图纸或原型,它定义了同类对象共有的变量和方法。例如,自行车类定义了自行车必须有的状态和行为:车轮、变速器、刹车器、如何驱动、如何变速、如何刹车等等。你可以用它生成一个有特定状态和方法的实例,别人也可以用它生成一个属于自己的实例。

对象和类的描述尽管十分相似,但它们还是有区别的。现实世界中,类不能代表它所描述的对象,这是非常清楚的,因为自行车图纸毕竟不是自行车。但在软件业中二者不太容易区分。一方面,由于软件对象都是现实对象或抽象概念的电子模型,另一方面,经常不加区分地将对象和类统称为“对象”。

Java 编程就是设计类,无论采用自定义方法还是继承方法设计一个类,最终是为了使用它的实例对象。

### 1.5.4 继承

OOP 允许由一个类定义另外一个类。例如山地车、赛车都属于自行车,它们是自行车的子类,换句话说,自行车是它们的父类。子类继承了父类的状态和行为,但并不局限于此。也就是说,子类可以增加新的变量和方法,有自己的特点。子类还可以覆盖(override)继承下来的方法,实现特殊要求。例如,你可以为山地车增加一个减速装置,并覆盖变速方法以便使用减速装置。

继承(inheritance)不但可以发生在同一个层次上,也可以发生在不同层次上。这种继承形成了一棵倒置的树,从根部开始发芽分支,长成一棵继承树。这棵树的根就是 Object,所有层次的类都是从 Object 类那里直接或间接地衍生下来的。Object 仅提供了所有的类在 Java 虚拟机上运行时所需要的基本属性和方法。一般来说,层次越高,类就越抽象,反之类就越具体。



继承使父类的代码得到重用,在继承父类提供的共同特性基础上添加新的代码,使编程不必一切都从头开始,有效提高了编程效率。

### 1.5.5 接口

接口(interface)可以看成是为两个不相关的实体提供交流途径的设备,例如语言就是两个人进行交流的接口。在 Java 中,接口就是为两个不相关的类提供交流的设备。接口非常类似于协议(protocol 一种共同行为的约定),是一个包含方法定义和常量值的集合。

Java 不支持多继承,子类只能有一个父类。有时需要使用其他类中的方法,但又无法直接继承,在这种情况下,只能使用接口技术。例如,Java 的事件单独放在一个类中,子类一般不直接从这个类派生。假如子类运行时需要响应各种事件,则需要使用事件类的处理方法。子类显然不能既从父类继承又从事件类继承,以便同时使用二者的方法,Java 的接口就是为了照顾这种情况。

接口不需要建立继承关系,就可以使两个不相关的类进行交互。接口提取了类的某些共同点,声明一些能被多个类实现的方法,但不给出方法体。接口由类的声明语句中的 implements 关键字引入,并在类体中实现接口的方法。例 1.2 和例 1.5 都使用了接口。

## 习 题

- 1-1 Java 有何特点?
- 1-2 Java 编译器产生的文件扩展名是什么?
- 1-3 运行 Java Applet 使用哪一个命令? 它能运行 Applet 的字节码文件吗?
- 1-4 运行 Java Application 有什么注意事项?
- 1-5 按照 Java 程序开发步骤,将例 1.2 编译并运行。
- 1-6 引入系统类有何作用?
- 1-7 Java Applet 和 Application 的运行方式有哪些不同?
- 1-8 简述对象、类、子类、消息的概念和它们之间的关系。
- 1-9 简述抽象、封装、继承的概念。
- 1-10 面向对象编程和面向过程编程有哪些不同?
- 1-11 编写一个 Java Application,输出字符串“I can write a java application!”。
- 1-12 编写一个 Java Applet,命名为 MyFirstApplet.java。要求输入一个名字“张华”,并显示字符串“张华,Java 欢迎你”。
- 1-13 编写一个 HTML 文件,将编译后的 MyFirstApplet.class 嵌入进去,然后用浏览器查看运行结果。
- 1-14 将例 1.2 改写为可进行四则算术运算的 Applet。



# 第2章

## Java 基本语法

本章的目的在于让读者对数据类型、表达式等 Java 基本语法内容有一个了解,这些内容是任何一门程序设计语言都必须包含的部分,也是我们编程的基础。

如果你已经熟悉了一种编程语言,特别是 C++ 的话,本章的内容会很熟悉,除了在某些方面 Java 语言更偏重于面向对象外,几乎所有的基本内容都与 C++ 相同或者类似。如果你是第一次学习编程语言,就要仔细阅读本章。虽然内容比较简单,但仔细弄清楚每一个环节,会使以后设计的程序更为成熟与合理。

### 2.1 Java 语言的基本组成

Java 语言主要由 5 种元素组成:标识符、关键字、直接量、运算符和分隔符。这 5 种元素有着不同的语法含义和组成规则,它们互相配合,共同完成 Java 的语意表达。本节首先介绍标识符、关键字和分隔符,直接量和运算符在下一节介绍。

#### 2.1.1 标识符

变量以及后面将要讲到的类和方法都需要一定的名称,这种名称就叫做标识符。什么是一个有效的标识符呢?在 Java 中,所有的标识符都必须以一个字母、下划线或美元符号“\$”开头。后面可以包含字母、数字、下划线和美元符号。

以上只是标识符命名的基本规则,表 2.1 是标识符命名正误对照表,通过它会对标识符的命名规则有一个具体的了解。

表 2.1 标识符命名正误对照表

合法标识符	非法标识符
MyClass	class
anInt	int
group7	7group
!	2
ONE_HUNDRED	ONE-HUNDRED

标识符是由用户自己规定的名称,可按上面的规则随意选取。不过,Java 中有一个标识符命名约定:常量用大写字母,变量用小写字母开始,类以大写字母开始。如果一个变量名由多个单词构成,第一个单词后面的单词以大写字母开始,例如 `anInt`。下划线虽然可以作为标识符的一员,但常用于常量名的单词分隔,因为常量名都是以大写字母单词命名的。还要注意一点,Java 严格区分字母大小写,标识符中的大小写字母被认为是不同的两个字符。例如以下是 4 个不同的合法标识符, `ad`, `Ad`, `aD`, `Da`。

## 2.1.2 关键字

关键字是 Java 语言本身使用的标识符,它有其特定的语法含义。所有的 Java 关键字将不能被用作标识符,如: `for`、`while`、`boolean` 等都是 Java 语言的关键字。关键字用英文小写字母表示,表 2.2 是一些常见的 Java 关键字。

表 2.2 Java 关键字

<code>abstract</code>	<code>default</code>	<code>goto *</code>	<code>null</code>	<code>switch</code>
<code>boolean</code>	<code>do</code>	<code>if</code>	<code>package</code>	<code>synchronized</code>
<code>break</code>	<code>double</code>	<code>implements</code>	<code>private</code>	<code>this</code>
<code>byte</code>	<code>else</code>	<code>import</code>	<code>protected</code>	<code>throw / throws</code>
<code>case</code>	<code>extends</code>	<code>instanceof</code>	<code>public</code>	<code>transient</code>
<code>catch</code>	<code>false</code>	<code>int</code>	<code>return</code>	<code>true</code>
<code>char</code>	<code>final</code>	<code>interface</code>	<code>short</code>	<code>try</code>
<code>class</code>	<code>finally</code>	<code>long</code>	<code>static</code>	<code>void</code>
<code>const *</code>	<code>float</code>	<code>native</code>	<code>strictfp</code>	<code>volatile</code>
<code>continue</code>	<code>for</code>	<code>new</code>	<code>super</code>	<code>while</code>

注:带 \* 号的关键字现在已不使用。

## 2.1.3 分隔符

分隔符是用来区分源程序中的基本成分,可使编译器确认代码在何处分隔。分隔符有注释、空白符和普通分隔符三种。

### 1. 注释

注释是程序员为了提高程序的可读性和可理解性,在源程序的开始或中间对程序的功能、作者、使用方法等所写的注解。注释仅用于阅读源程序,系统编译程序时,忽略其中的所有注释。注释有两种类型:

(1) 注释一行,以“`//`”开始,最后以回车结束。一般作单行注释使用,也可放在某个语句的后面;

(2) 一行或多行注释,以“`/*`”开始,最后以“`*/`”结束,中间可写多行。

## 2. 空白符

空白符包括空格、回车、换行和制表符(Tab 键)等符号,用来作为程序中各种基本成分之间的分隔符。各基本成分之间可以有一个或多个空白符,其作用相同。和注释一样,系统编译程序时,只用空白符区分各种基本成分,然后忽略它。

## 3. 普通分隔符

普通分隔符和空白符的作用相同,用来区分程序中的各种基本成分,但它在程序中有确定的含义,不能忽略。Java 有 4 种普通分隔符:

- { } 大括号,用来定义复合语句、方法体、类体及数组的初始化;
- ; 分号,是语句结束的标志;
- , 逗号,分隔方法的参数和变量说明等;
- : 冒号,说明语句标号。

**例 2.1** 标识符、关键字和分隔符的使用。

```
public class Example {  
    public static void main(String args[]) {  
        int i, c;  
        ...  
    }  
}
```

程序说明: class 是关键字,用来定义类, public 说明这个类是公有的。Example 是标识符,用来说明类的名称。为了与类的其他成员区别,一般将类名的第一个字母大写。

第一行末尾与最后一行是一对大括号,括号中为类体。类体中定义了类的成员: 属性和方法,并且指明成员的访问权限,本例中只定义了一个方法 main。

第二行 public static void 为关键字,说明方法 main 是公有的(public),是类的静态成员(static),无返回值(void)。需要注意的是,Java Application 中 main 方法的声明必须采用这种方式。括号里的内容是参数定义, String 是字符串类名, args 是数组名。

方法体中的 int 是关键字,说明 i 与 c 为整型变量。

## 2.2 数据类型

数据类型说明了常量、变量或表达式的性质。Java 的数据类型可分为:

- 基本类型 包括整型、浮点型、布尔型和字符型;
- 数组类型 包括一维数组和多维数组;
- 复合类型 包括类、接口。

### 2.2.1 基本数据类型

表 2.3 给出了 Java 基本数据类型。Java 定义了 8 种基本数据类型,利用基本数据类

型可以构造出复杂数据结构来满足各种需要。Java 是严格区分数据类型的语言,要求在程序中使用任何变量之前必须声明其类型。和 C++ 不同的是,Java 的基本数据类型长度是固定的。例如 int 类型在任何计算机上的长度都是 32 位长(4 个字节),这就使得 Java 的基本数据类型可以跨平台自由移植。Java 的 char 类型采用了国际编码标准 Unicode,每个码有 16 位长(2 个字节),可容纳 65 536 个字符,有效地解决了用 ASCII 双字节码表示东方文字带来的诸多不便,使 Java 处理多语种的能力大大加强。

表 2.3 Java 基本数据类型

数据类型	名称	位长	默认值	取值范围
布尔型	boolean	1	false	true, false
字节型	byte	8	0	-128~127
字符型	char	16	'\u0000'	'\u0000'~'\uffff'
短整型	short	16	0	-32 768~32 767
整型	int	32	0	-2,147,483,648~2,147,483,647
长整型	long	64	0	-9 223,372,036,854,775,808~ 9 223,372,036,854,775,807
浮点型	float	32	0.0	1.4E-45~3.402 823 5E+38
双精度型	double	64	0.0	4.9E-324~1.797 693 134 862 315 7E+308

## 2.2.2 直接量

直接量(literal)是指在程序中直接给出的一个符号串,作用是为变量赋值或参与表达式运算。直接量可以是一个具体的数值或字符串,也称常量。但 Java 中的常量另有所指,是用 final 说明的一个标识符,在很多教科书中往往不加区分地统称为常量。将一个标识符说明为常量,然后赋予它一个直接量,就在二者之间划上了等号。直接量或常量在程序执行过程中是不可更改的,它们与变量的区别是不占用内存。

### 1. 布尔常量

布尔常量只有两个值 true 和 false,代表了两种状态:真和假,书写时直接使用这两个单词,不能加引号。

### 2. 整型常量

整型常量是不含小数的整数值,书写时可采用十进制、十六进制和八进制形式。十进制常量以非 0 开头,八进制以 0 开头,十六进制则以 0X 开头。下面是三条赋值语句:

```
int i=15;
int j=017;
int k=0xF;
```

赋值后,三个变量相等,值的大小都是十进制的 15。整型常量默认为 32 位的 int 型,

如果在数值后边加上 L 或 l,则表示为 64 位的长整型。

### 3. 浮点型常量

Java 的浮点型常量有两种表示形式:

- (1) 十进制数形式,由数字和小数点组成,且必须有小数点,如. 123,0. 123,123. 0;
- (2) 科学计数法形式,如 123e3 或 123E-3,其中 e 或 E 之前必须有数,且 e 或 E 后面的指数必须为整数。

对于一个浮点数,加上 f 或 F 后缀,就是单精度浮点数;加上 d 或 D 后缀,就是双精度浮点数。不加后缀的浮点数被默认为双精度浮点数,双精度浮点数在计算机中占 64 位,有很高的精度。

### 4. 字符常量

字符常量是由一对单引号括起来的单个字符。它可以是 Unicode 字符集中的任意一个字符,如:'a','Z'。对无法通过键盘输入的字符,可用转义符号表示,见表 2.4。

字符常量的另外一种表示就是直接写出字符编码,如字母 A 的八进制表示为'\101',十六进制表示为'\u0041'。

表 2.4 转义符号表

转义符号	Unicode 编码	功 能
'\b'	'\u0008'	退格
'\r'	'\u000d'	回车
'\n'	'\u000a'	换行
'\t'	'\u0009'	水平制表符
'\f'	'\u000c'	进纸
'\''	'\u0027'	单引号
'\"'	'\u0022'	双引号
'\\'	'\u005c'	反斜杠

### 5. 字符串常量

字符串常量是用一对双引号括起来的字符序列。当字符串只包含一个字符时,不要把它和字符常量混淆,例如'A'是字符常量,而"A"是字符串常量。字符串常量中可包含转义字符,例如"Hello\n world!"在中间加入了一个换行符,输出时,这两个单词将显示在两行上。

## 2.2.3 变量

变量是内存中的一块空间,提供了可以存放信息和数据的地方,具有记忆数据的功能。变量是可以改变的,它可以存放不同类型的数据,通常用字母或单词作为变量名。

## 1. 变量的声明

在 Java 中存储一个数据,必须将它保存到一个变量中。变量在使用前必须有定义,即有确定的类型和名称。声明变量的语句如下:

类型 变量名[,变量名][=初值];

该语句告诉编译器以给定的数据类型和变量名建立的一个变量。可以一次声明多个变量,并同时赋初值。例如:

```
byte b1, b2;  
int v1=0, v2=10, v3=18;
```

上面的两条语句声明了 2 个字节型变量 b1、b2,3 个整型变量 v1、v2、v3,分别赋值 0、10 和 18。

**例 2.2** 变量声明示例。使用 JDK 编译并运行该程序,结果如图 2.1 所示。

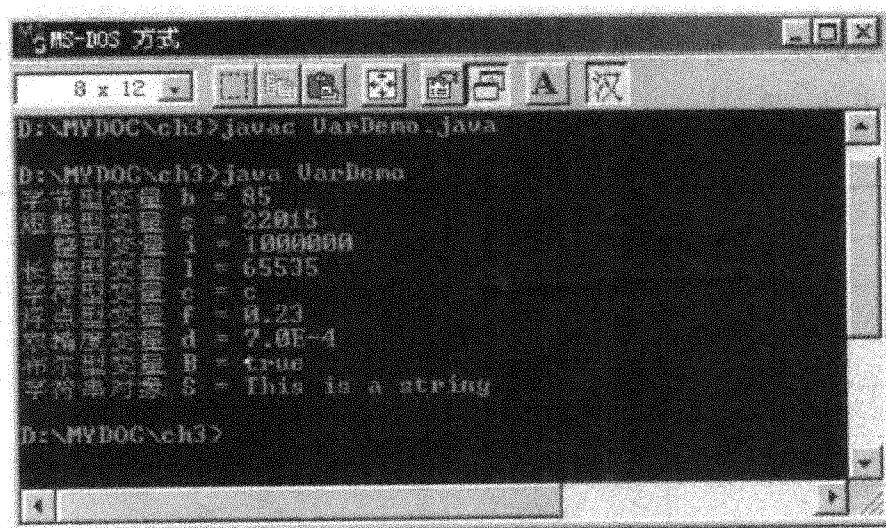


图 2.1

```
public class VarDemo {  
    public static void main(String args[]) {  
        byte b=0x55;  
        short s=0x55ff;  
        int i=1000000;  
        long l=0xffffL;  
        char c='c';  
        float f=0.23F;  
        double d=0.7E-3;  
        boolean B=true;  
        String S="This is a string";  
  
        System.out.println("字节型变量 b = "+b);
```



```

        System.out.println("短整型变量 s = "+s);
        System.out.println(" 整型变量 i = "+i);
        System.out.println("长整型变量 l = "+l);
        System.out.println("字符型变量 c = "+c);
        System.out.println("浮点型变量 f = "+f);
        System.out.println("双精度变量 d = "+d);
        System.out.println("布尔型变量 B = "+B);
        System.out.println("字符串对象 S = "+S);
    }
}

```

## 2. 变量的使用范围

当你声明了一个变量后,它将被引入到一个范围当中。也就是说,该变量只能在程序的特定范围内使用,出了这个范围,变量就消失了。

在类中声明的变量是成员变量,通常在类的开始处声明。在方法和块中声明的变量叫局部变量,使用范围是从它被声明的地方到它所在那个块的结束处,块是由两个大括号所定义的,见例 2.3 中的程序。

**例 2.3** 变量的使用范围。

```

class Example {
    int i;
    public static void main(String args[]) {
        int j;
        ...
        {
            if (条件) int k=10;
        }
        System.out.println("k="+k); // 编译时将出错,已出 k 的使用范围
    }
}

```

k 在一个块中声明,在这个块之外它是不存在的,所以编译时会出错。

## 3. 变量类型的转换

系统方法 `System.in.read` 返回一个整型数值,但你却常常想要把它当作一个字符来使用。现在的问题是,当有一个整数而你需要把它变成一个字符时应当做些什么呢?你需要去做的是将一个整数类型转换为一个字符型。变量类型的转换可采用在表达式前加上类型转换符的方式实现:(类型)表达式。例如:

```

int a;
char b;

```

```
b='A';  
a=(int) b;
```

上面的语句表示 a 为整型, b 为字符型, (int) 告诉编译器你想把字符型的 b 变成整型并把它放在变量 a 里。

记住整型和字符型变量位长的不同是非常重要的, 整型是 32 位, 而字符型是 16 位, 所以从整型转换到字符型可能会丢失信息。

同样, 当你把 64 位的长整型数转换为整型数时, 由于长整型有比 32 位更多的信息, 也很可能会丢失信息。即使两个数有相同的位数, 比如整型和浮点型(都是 32 位), 从浮点型转换为整形时也会丢失信息。

Java 不允许自动类型转换, 当你进行类型转换时, 要注意使目标类型能够容纳原类型的所有信息。

表 2.5 列出了不会丢失信息的类型转换。

表 2.5 不会丢失信息的类型转换

原始类型	目标类型
byte	short, char, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

## 2.3 运算符与表达式

Java 的运算符代表着特定的运算指令, 程序运行时将对运算符连接的操作数进行相应的运算。运算符和操作数的组合形成了表达式, 表达式代表着一个确定的量。表达式在使用上总是先计算后使用, 因此, 不管一个表达式有多么复杂, 其最终结果仍是一个有确定类型和大小的量。

按照运算符功能来分, 运算符有 7 种: 赋值运算符、算术运算符、关系运算符、条件运算符、逻辑运算符、位运算符和其他运算符。

按照连接操作数的多少来分, 有一元运算符、二元运算符和三元运算符。另外, 算术运算符和赋值运算符可以结合在一起形成算术赋值运算符。

### 2.3.1 赋值运算符

表 2.6 给出了赋值运算符和功能说明(未列出包含位运算符的赋值运算符)。

表 2.6 赋值运算符

运算符	功 能	举 例	等价于
=	右边数赋给左边变量	x=5, 5 赋给 x	
+=	左右两边数相加, 结果赋给左变量	x=5, x+=10 将 15 赋给 x	x=x+10
-=	左右两边数相减, 结果赋给左变量	x=5, x-=10 将 -5 赋给 x	x=x-10
*=	左右两边数相乘, 结果赋给左变量	x=5, x*=10 将 50 赋给 x	x=x*10
/=	右边数除左边数, 结果赋给左变量	x=5, x/=5 将 1 赋给 x	x=x/5
%=	右边数除左边数, 余数赋给左变量	x=25, x%=7 将 4 赋给 x	x=x%7

由赋值运算符构成的表达式称为赋值表达式。赋值运算符的右边是一个表达式, 这个表达式还可以包含一个表达式。例如:  $a=b=c=0$ , 相当于三个表达式  $c=0$ ,  $b=c$ ,  $a=b$ 。

### 2.3.2 算术运算符

表 2.7 给出了算术运算符和功能说明。

表 2.7 算术运算符

运算符	功 能	举 例	说 明
-	取负	-x	单目运算, 将 x 取负值
++	加 1	i++, ++i	单目运算, 等价于 $i = i + 1$
--	减 1	i--, --i	单目运算, 等价于 $i = i - 1$
*	乘	$5 * 3 \rightarrow 15$	
/	除	$5 / 3 \rightarrow 1$	整数相除取商的整数部分
%	求余	$5 \% 3 \rightarrow 2$	两数相除取余数
+	加	$5 + 3 \rightarrow 8$	
-	减	$5 - 3 \rightarrow 2$	

算术运算符按操作数的多少可分为一元运算符和二元运算符, 一元运算符一次对一个操作数进行运算, 二元运算符一次对两个操作数进行运算。对于表达式的运算来说, 如果有一个变量或操作数是长整型的, 那么结果就肯定是长整型的, 否则即使操作数还没有确定是字节型、短整型或字符型, 运算结果都是整型。浮点型和双精度型的情况是类似的。

求余运算在判断两个数是否成倍数时很有用, 余数为 0 则两数成倍数, 否则两数不成倍数。

$i++$  和  $i--$  比传统写法的加减运算速度要快很多, 常用作循环结构中的计数器。加 1 或减 1 的书写位置对运算结果有很大影响, 例如:

```
int i=10;
int j=10;
```

```
int x=i++;
int y=++j;
```

则 x 的值为 10, y 的值为 11, i 和 j 的值均为 11。读者可能要问 x 和 y 的值不是应该为 11 吗? 原因在于“++”出现在操作数左边时, 先对操作数加 1 然后再使用; 出现在操作数右边时, 则先使用操作数然后再加 1。无论“++”出现在什么位置上, 对操作数的最终取值没有影响, 但对表达式的值有直接影响。“--”与之类似。

### 2.3.3 关系运算符

有一些运算符能产生布尔类型的结果, 我们把它称为关系运算符, 表 2.8 列出了这些运算符。

表 2.8 关系运算符

运算符	含 义	举 例	说 明
>	大于	'A' > 'a' → false	a 的编码值大于 A 的编码值
<	小于	'A' < 'a' → true	A 的编码值小于 a 的编码值
>=	大于等于	5 >= 3 → true	
<=	大于等于	5 <= 3 → false	
==	等于	5 == 3 → false	
!=	不等于	(3+3) != 5 → true	先计算 3+3 的值再比较

关系运算符用于两个操作数之间关系的比较。关系表达式的运算结果为布尔值, 不是 true 就是 false, 操作数可以是常量、变量和表达式。关系表达式常常用作分支结构或循环结构的控制条件。

注意: Java 的相等运算符“==”可能会给你带来麻烦, 很多人在比较两个量时往往错用了等号“=”, 以至于变成了赋值操作, 编程中一定要注意比较两个量时用“==”。

### 2.3.4 条件运算符

条件运算符有一个“?”和一个“:”, 条件运算符与上面的运算符略有不同, 是三元运算符。条件表达式的结构为:

(条件) ? 结果 1: 结果 2;

条件表达式的计算过程为: 首先计算作为条件的逻辑表达式或关系表达式, 返回值为 true 时表达式的值为结果 1, 返回值是 false 时表达式的值为结果 2。

条件表达式可取代简单的二分支结构, 书写简单, 并有较快的运算速度。例如:

```
int a=5, b=2, result;
if (a>b)
    result=a-b;
else
    result=b-a;
```

其中的分支结构可改写为: `result = a > b ? a - b : b - a;`

### 2.3.5 逻辑运算符

表 2.9 列出了逻辑运算符。

表 2.9 逻辑运算符

运算符	含 义	举 例	说 明
!	逻辑非	<code>!(3 &gt;= 5) → true</code>	将表达式的值真变假,假变真
&&	逻辑与	<code>(3 &lt; 5) &amp;&amp; (6 &gt; 4) → true</code>	仅当两个表达式都为真时才为真
	逻辑或	<code>(3 &gt; 5)    (6 &lt; 4) → false</code>	仅当两个表达式都为假时才为假
^	逻辑异或	<code>(3 &lt; 5) ^ (6 &lt; 4) → true</code>	仅当两个表达式值相异时才为真
&	布尔逻辑与	同 &&	在表达式计算上与 && 不同
	布尔逻辑与	同	在表达式计算上与    不同

关系运算只能解决一些简单条件的判定问题,对较为复杂的条件可用逻辑运算来判定。逻辑表达式通常由多个关系表达式构成,最终运算结果为布尔值 `true` 或 `false`。

“&&”连接的两个表达式中,只要有一个不为真,则整个表达式就为假。运算时只要判定左边表达式为假,就可立即得出结论,不再计算右边表达式。所以,最有可能取假值的表达式尽量放在左边,这样可提高表达式的运算速度。

“&”在表达式判定上和“&&”相同,唯一不同的是它总是计算两边表达式的值。

“||”连接的两个表达式中,只要有一个为真,则整个表达式就为真。运算时只要判定左边表达式为真,就可立即得出结论,不再计算右边表达式。所以,最有可能取真值的表达式尽量放在左边,以提高表达式的运算速度。

“|”在表达式判定上和“||”相同,不同之处是它总是计算两边表达式的值。

“^”连接的两个表达式同为真或同为假时,整个表达式为假,其他情况下取真值。

**例 2.4** 下面的例子说明了关系运算符和逻辑运算符的使用。输出结果如图 2.2 所示。

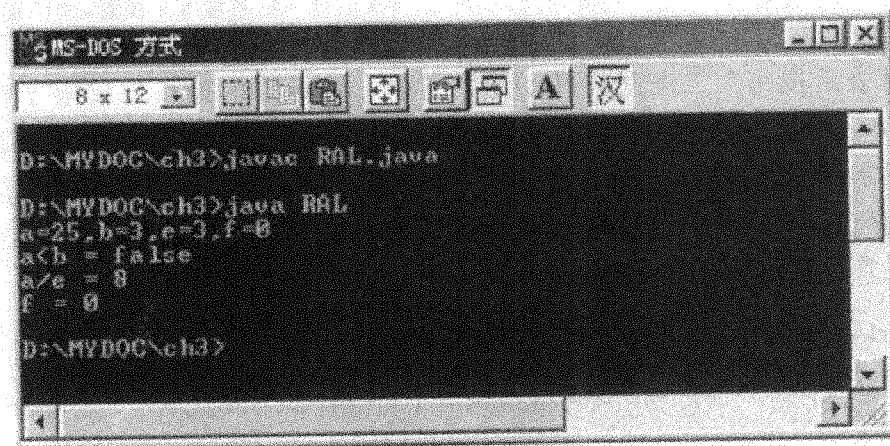


图 2.2

```

public class RAL {
    public static void main(String args[]) {
        int a=25, b=3, e=3, f=0;
        boolean d=a<b;

        System.out.println("a=25,b=3,e=3,f=0");
        System.out.println("a<b = "+d);

        if (e!=0 && a/e>5)
            System.out.println("a/e = "+a/e);
        if (f!=0 && a/f>5)
            System.out.println("a/f = "+a/f);
        else
            System.out.println("f = "+f);
    }
}

```

## 2.3.6 位运算符

表 2.10 列出了位运算符。

表 2.10 位运算符

运算符	含 义	举 例	说 明
~	位非	~ a	对 a 的每一位求反,即 1 变 0,0 变 1
&	位与	a & b	将 a 和 b 的对应位进行与运算
	位或	a   b	将 a 和 b 的对应位进行或运算
^	位异或	a ^ b	将 a 和 b 的对应位进行异或运算
<<	位左移	a << b	a 的每一位左移 b 个位置
>>	位右移	a >> b	a 的每一位右移 b 个位置
>>>	位右移	a >>> b	a 的每一位右移 b 个位置,原位置以 0 填充

位运算表示按每个二进制位(bit)进行计算。位运算的操作数仅限于整数(char、short、int、long),以二进制形式进行,运算结果为一个整数。位运算主要是为了满足编程中的特殊要求,如测试设备、与硬件通讯等高级编程内容。一般计算机中 8 个位为 1 个字节,数据保存在 1 个或多个字节中,所以位运算要求程序员必须熟悉数的二进制表示,并且知道位运算以后的结果是什么。例如,

```

char a=14, b=3;
char c=a&b;
char d=a<<b;

```

位与运算过程:

00001110	a=14
& 00000011	b=3
-----	
00000010	c=2

位左移运算过程:

00001110	<< 3	01110000
a=14		d=112

a 值 14 的二进制表示为 00001110(省去高位 3 个字节),b 值 3 为 00000011。进行位与运算时,a 和 b 对应位上如果都是 1,则结果 c 的对应位上为 1,否则为 0,运算后 c 值为 2。左移运算时,“111”向左边移动 3 位,d 值变为“01110000”,对应的十进制就是 112。

细心的读者会发现,位左移运算相当于乘法( $14 * 2 * 2 * 2 = 112$ ),确实是这样。位左移 1 位相当于原数乘以 2,位右移 1 位相当于原数除以 2。

“>>”与“>>>”的基本功能都是右移,但“>>”可保持符号位不变,而“>>>”则是用零来填充右移后所留下的空位,包括符号位。

对一个正整数,例如 8, $8 >> 1$  和  $8 >>> 1$  的效果相同,8 的二进制数为 00001000,右移 1 位后为 00000100,都是 4。

对于一个负数,其最高位(符号位)是 1,进行“>>”运算时,符号位不变,只移动后面的数,例如 -4 的二进制数为 [1]0100,[ ] 中的 1 表示符号位(1 代表负数), $-4 >> 1$  的结果为 [1]0010,即为 -2;如果对负数进行 >>> 运算时,移动后左边的空位(包括符号位)置 0,负数变成正数, $-4 >>> 1$  的结果 [0]0010,即为 2。

**例 2.5** 下面的例子将整形数转为二进制输出,结果如图 2.3 所示。

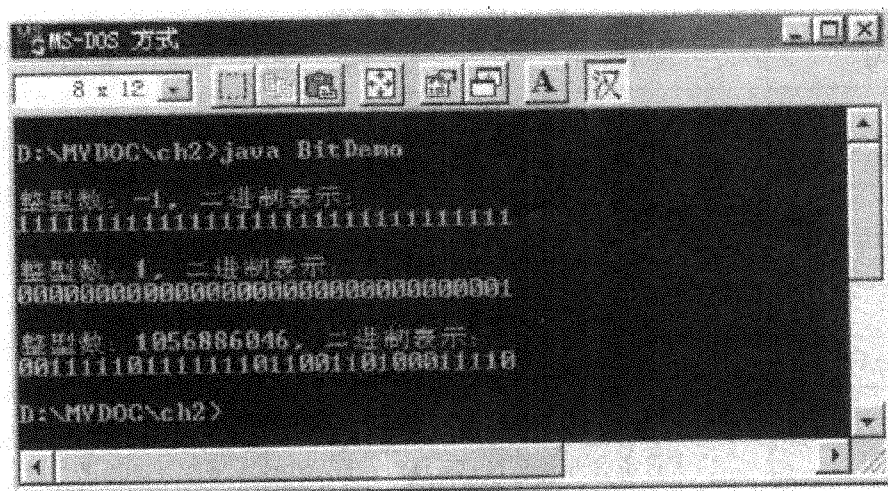


图 2.3

```
import java.util.*;
public class BitDemo {
    public static void main(String[] args) {
        Random rand=new Random();
```

```

        toBin(-1);
        toBin(+1);
        toBin(rand. nextInt());
    }

    static void toBin(int x) {
        System.out.println("\n 整型数:" + x + ", 二进制表示:");
        for(int i=31; i>=0; i--)
            if (((1<<i) & x) != 0)
                System.out.print("1");
            else
                System.out.print("0");
        System.out.print("\n");
    }
}

```

程序中 toBin 是转换方法,将主程序传递过来的整型数转为二进制数输出。一个整型数有 32 位,用一个循环逐个判断整型数的每一位上是否为 1,如为 1 则输出字符“1”,否则输出“0”,判断是用 1 右移到某位上然后和整型数的对应位做“&”运算。

主程序首先传递了-1 和+1,然后使用随机数发生器产生一个随机整数并传递给转换方法 toBin 输出。

### 2.3.7 其他运算符

以上是常见的分类运算符,在 Java 中还有一些特殊的运算符,如表 2.11。

表 2.11 其他运算符

符 号	功 能
( )	表达式加括号则优先运算
(参数表)	方法的参数传递,多个参数时用逗号分隔
(类型)	强制类型转换
.	分量运算符,用于对象属性或方法的引用
[]	下标运算符,引用数组元素
instanceof	对象运算符,用于测试一个对象是否是一个指定类的实例
new	对象实例化运算符,实例化一个对象,即为对象分配内存
+	字符串合并运算符,如“Hello”+“ World!”等于“Hello World!”

### 2.3.8 运算符的优先级

知道了许多运算符,那么当表达式里有多种运算符号的时候,运算的次序是什么呢?表 2.12 从高到低列出了运算符的优先级,同一行中的运算符优先级相同。

优先级是指同一式子中多个运算符被执行的次序。同一级别里的运算符具有相同的



优先级,算术运算符具有左结合性,例如计算  $a-b+c$  时,  $b$  先与左边的减号结合,执行  $a-b$  的运算,再执行加  $c$  的运算。赋值运算符具有右结合特性,当计算  $a=b=c=0$  时,先执行  $c=0$ ,再执行  $b=c$ ,最后执行  $a=b$ 。

对于表达式  $a=b+c*d/(e<<f)$ ,Java 处理时会按照表 2.12 的次序进行先高后低的计算。先计算括号内的  $e<<f$ ,接着是  $c*d$ ,然后除以  $e<<f$  的值,最后把上述结果与  $b$  的和存储到变量  $a$  中。

不论任何时候,当你一时无法确定某种计算的执行次序时,可以使用加括号的方法明确为编译器指定运算顺序,这也是提高程序可读性的一个重要方法。

表 2.12 运算符的优先级

1	.	[]	()	expr++	expr--
2	++expr	--expr	!	~	-
3	new	(type)			
4	*	/	%		
5	+	-			
6	<<	>>	>>>		
7	<	>	<=	>=	instanceof
8	==	!=			
9	&				
10	-				
11					
12	&&				
13					
14	?:				
15	=	opr=			

## 2.4 数组

数组是有序数据的集合,数组中的每个元素具有相同的数组名,根据数组名和下标来唯一地确定数组中的元素。数组有一维数组和多维数组,使用时要先声明后创建。

### 2.4.1 一维数组的声明

数组是 Java 语言中的特殊数据类型,它们保存着能通过下标索引来引用的一组同类数据。一维数组是指一个线性数据序列,声明方式为:

数据类型 数组名[]; 或 数据类型[] 数组名;

例如：

```
int arr1[];  
char [] arr2;
```

声明了两个数组，arr1 是整型数组，arr2 是字符型数组。

说明：其中“数据类型”可以是 Java 中任意的数据类型，“数组名”为一个合法的标识符，“[]”指明该变量是一个数组类型变量并且是一维的。

Java 在数组的定义中并不为数组元素分配内存，因此“[]”中不用指出数组中元素的个数即数组长度，而且对于如上定义的一个数组暂时还不能访问它的任何元素。

## 2.4.2 一维数组的创建与赋值

定义数组后，还必须为数组分配内存、初始化。这有两种方法。

### 1. 用运算符 new 分配内存再赋值

命令格式如下：数组名 = new 数据类型[size]

其中，size 指明数组的长度。如：

```
int intArr[] = new int[3];  
char chArr[] = new char[5];
```

上述语句将数组的声明和创建放在一起，第一条语句声明了一个整型数组并分配了 3 个整型数据所占据的内存空间，顺序为 intArr[0]，intArr[1]，intArr[2]。注意下标是从第一个元素的 0 开始，到数组长度减 1。第二条语句声明并分配了 5 个 char 型变量所占据的内存空间。

数组元素赋值的方法与变量相同，如：

```
intArr[0] = 10;  
intArr[1] = 20;  
intArr[2] = 30;
```

### 2. 直接赋初值并定义数组的大小

第二种方法是直接赋初值并定义数组的大小。初值必须用大括号括起，用逗号作分隔符，初值的个数表示数组的大小。例如：

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};  
char c[] = {'a', 'b', 'c', '北', '京'};
```

**例 2.6** 一维数组的使用。该程序对数组中的每个元素赋值，然后按逆序输出，结果如图 2.4 所示。

```
public class ArrDemo {  
    public static void main(String args[]) {  
        int i;
```

```

int a[]=new int[5];

for (i=0; i<5; i++)
    a[i]=i;

for (i=a.length-1; i>=0; i--)
    System.out.println("a["+i+"] = "+a[i]);
}
}

```



图 2.4

说明:数组作为 Array 类的实例,继承了 Array 类的属性和方法,求数组长度时可引用 Array 的 length 属性。

### 2.4.3 多维数组

与 C++ 一样,Java 将多维数组看作数组的数组。例如二维数组就是一个特殊的一维数组,它的每个元素是一个一维数组。下面我们主要以二维数组为例来进行说明,更高维的情况是类似的。

**例 2.7 二维数组的使用。**该程序对二维数组中的每个元素赋值,然后输出,程序运行结果如图 2.5 所示。

```

import java.applet.*;
import java.awt.*;

public class ArrDemo1 extends Applet {
    public void paint(Graphics g) {
        int arr1[][]=new int[3][4];
        int arr2[][]=new int[3][ ];
        int arr3[][]={{0,1,2},{3,4,5},{6,7,8}};
        int i,j,k=0;
        for (i=0; i<3; i++) // arr1 为 3 行 4 列
            for (j=0; j<4; j++)

```

```

        arr1[i][j]=k++;
    for (i=0; i<3; i++) // arr2 每一行是变长的,元素个数为 3、4、5
        arr2[i]=new int[i+3];
    for (i=0; i<3; i++) // 为 arr2 赋值
        for (j=0; j<arr2[i].length; j++)
            arr2[i][j]=k++;

    g.drawString(" arr1:",20,20);
    for (i=0; i<3; i++)
        for (j=0; j<4; j++)
            g.drawString(""+arr1[i][j],20+20*j,40+20*i);
    g.drawString(" arr2:",115,20);
    for (i=0; i<3; i++)
        for (j=0; j<arr2[i].length; j++)
            g.drawString(""+arr2[i][j],115+20*j,40+20*i);
    g.drawString(" arr3:",230,20);
    for (i=0; i<3; i++) // arr3 为 3 行 3 列
        for (j=0; j<3; j++)
            g.drawString(""+arr3[i][j],230+20*j,40+20*i);
    }
}

```

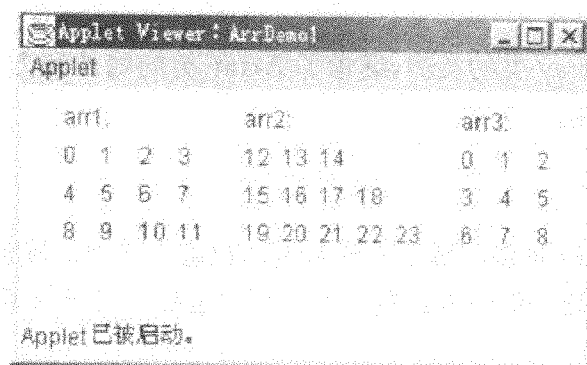


图 2.5

例 2.7 总结了二维数组的使用方法。二维数组的声明与一维数组基本相同,只是后面再加上一对“[]”。创建二维数组时,可指定各维的长度或至少指定第一维的长度,也可采用直接赋值的方法确定二维数组的长度。此时,按照给定的值序依次填满数组每一行中的元素。

二维数组有一个好处是第二维的长度可以不相等。例如程序中利用循环使 arr2 的第二维长度分别为 3、4、5,适用于数组元素不等的情况。

## 习 题

### 2-1 Java 有哪些基本数据类型?

2-2 float 和 double 型数据在赋值时有哪些注意事项?

2-3 Java 的字符常量和字符串常量有何区别?

2-4 数据类型转换有何作用?

2-5 说明 System.out.println("This character '"+A+"' has the value: "+(int)'A') 的输出结果。

2-6 说明表达式  $3 * 9 * (3 + (9 * 3 / (-3)))$  的计算次序。

2-7 计算表达式的值:  $x + a \% 3 * (int)(x + y) \% 2 / 4$ , 设  $x = 2.5, a = 7, y = 4.7$ 。

2-8 设 x 的值为 10, 写出表达式运算后 x 的值。

```
x += x
x -= 3
x *= 1 + 2
x %= 5
```

2-9 设  $a = 6, b = -4$ , 计算表达式的值。

```
--a%++b
(--a)<<a
(a<10 && a>10 ? a : b)
```

2-10 编写一个程序, 能计算输出 0 到 10 的平方和立方值。

2-11 Fibonacci 数列定义为:  $F_1 = F_2 = 1, F_n = F_{n-1} + F_{n-2} (n \geq 3)$ , 下面是输出 Fibonacci 数列的程序, 请写出输出结果。

```
public class Fib {
    public static void main(String args[]) {
        int i;
        int arr[] = new int[10];
        arr[0] = arr[1] = 1;
        for(i = 2; i < 10; i++)
            arr[i] = arr[i-1] + arr[i-2];
        for(i = 1; i <= 10; i++)
            System.out.println("F["+i+"]="+arr[i-1]);
    }
}
```



# 第3章

## Java 语句及其控制结构

Java 程序是语句的集合,如何组织 Java 语句使它们能够顺利实现程序的功能呢?学完本章的内容,你就知道:Java 用类搭起程序的框架,以方法实现类的功能,在方法中用各种语句结构来控制程序的流程。

### 3.1 Java 程序结构

#### 3.1.1 Java 程序构成

**例 3.1** 用一个程序范例,说明 Java 程序的构成。

```
import java.applet.Applet;
import java.awt.*;
public class GetSquare extends Applet
{
    Label label1;
    public void init() {
        label1 = new Label("前 30 个数的平方");
        add(label1);
    }
    public void paint(Graphics g) {
        for (int i = 0; i < 30; i++) {
            int x=i%10, y=i/10;
            g.drawString(String.valueOf((i+1) * (i+1)),
                20+30 * x, 50+25 * y);
        }
    }
}
```

Java 包引入语句  
— 类声明语句  
— 成员变量  
init 方法  
循环  
paint 方法  
类体

Java 源程序一般包括两部分:Java 包引入(如果有的话)部分和类定义部分。其中类

定义部分是必不可少的。一个源程序可以定义多个类,但只有一个主类。Java Applet 的主类就是继承自 Applet 的子类,Java Application 的主类就是包含 main 方法的类。

类定义由类说明语句和类体组成。类体也由两部分组成:成员变量和成员方法。在上例 GetSquare 类中, label1 是添加的成员变量(又称属性、域),init 和 paint 是继承下来的成员方法。你应该注意到,类体中除了成员变量说明语句外没有其他语句,Java 规定所有操作性语句必须放在成员方法中。

方法类似于 C++ 的函数,除了可以继承父类的方法外,你也可以加入自定义的成员方法。根据需要,方法中可以定义变量,但更重要的是组织语句。Java 程序正是由各种语句结构控制着程序流程和功能的实现。

### 3.1.2 Java 语句

Java 语句是 Java 标识符的集合,由关键字、常量、变量和表达式构成,是成员方法的主要成分,必须包含在类的方法体之中。Java 语句有表达式语句、复合语句、选择语句和循环语句等。语句以分号“;”作为结束标志,单独的一个分号被看作一个空语句,空语句不做任何事情。

在表达式后边加上分号“;”,就是一个表达式语句。经常使用的表达式语句有赋值语句和方法调用语句。表达式语句是最简单的语句,它们被顺序执行,完成相应的操作。

复合语句也称为块(block)语句,是包含在一对大括号“{}”中的任意语句序列。与其他语句用分号作结束符不同,复合语句右括号“}”后面不需要分号。尽管复合语句含有任意多个语句,但从语法上讲,一个复合语句被看作一个简单语句。

#### 例 3.2 复合语句示例。

```
class Block {  
    public static void main(String args[]) {  
        int k, i=3, j=4;  
        k=i+j;  
        System.out.println("k="+k);  
        {  
            float f;  
            f=j+4.5F;  
            i++;  
            System.out.println("f="+f);  
        }  
        System.out.println("i="+i);  
    }  
}
```

程序说明:在 main 方法中有两个复合语句嵌套在一起,复合语句内包含的是表达式语句。第一个复合语句中说明了三个整型变量 k、i、j,它们不仅在第一个复合语句中起作用,还在被嵌套的第二个复合语句中起作用。而在第二个复合语句中说明的变量 f 仅在第二个复合语句中起作用。



在这个例子中,人为地加入了一个复合语句,在实际编程中并不多见。复合语句更广泛的应用是在结构式语句中,如选择语句和循环语句。当结构中包含的表达式语句超过一条时,就要用大括号把它们括起来。

## 3.2 选择语句

在复合语句中必须逐行执行每条命令。能否改变程序执行的顺序呢?利用 if... else 结构就可以根据条件控制程序流程。

### 3.2.1 if 语句

if 语句的语法结构如下:

```
if (条件表达式)
    s1 语句;
```

这是最简单的单分支结构。条件表达式的值为 true,就执行 s1 语句,否则就忽略 s1 语句。s1 语句可以是复合语句。

### 3.2.2 if... else 语句

if 语句通常都与 else 语句配套使用,形成二分支结构。它的语法结构如下:

```
if (条件表达式)
    s1 语句;
else
    s2 语句;
```

当条件表达式的值为 true,就执行 s1 语句,忽略 else 和 s2 语句;否则,条件表达式的值为 false,程序忽略 s1 语句,执行 else 后面的 s2 语句。s1 和 s2 都可以是复合语句。

**例 3.3** 比较两个数的大小并按升序输出,结果如图 3.1 所示。



图 3.1

```
class Compare {
    public static void main(String args[]) {
        double d1=23.4;
```

```

double d2=35.1;
if(d2>=d1)
    System.out.println(d2+">="+d1);
else
    System.out.println(d1+">="+d2);
}
}

```

### 3.2.3 if...else 复合结构

对于复杂的情况,我们可以嵌套使用 if...else 语句。它的语法结构如下:

```

if (条件表达式 1)
    s1 语句;
else if (条件表达式 2)
    s2 语句;
else
    s3 语句;

```

在这里依次计算条件表达式,如果某个条件表达式的值为 true,就执行它后面的语句,其余部分被忽略;所有表达式的值都为 false,就执行最后一个 else 后的 s3 语句。s1、s2 和 s3 都可以是复合语句。

**例 3.4** 下面是一个用 if...else 语句构造多分支程序的例子,判断某一年是否为闰年。

闰年的条件是符合下面二者之一:能被 4 整除,但不能被 100 整除;能被 4 整除,又能被 100 整除。输出结果如图 3.2 所示。

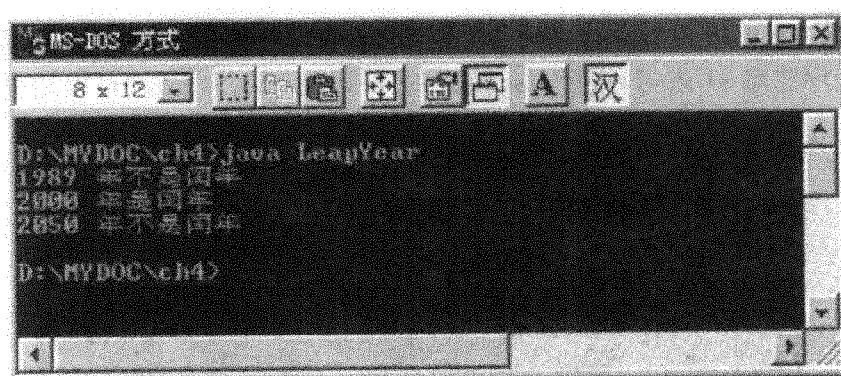


图 3.2

```

public class LeapYear {
    public static void main(String args[]) {
        boolean leap;
        int year=1989;

        if ((year%4==0 && year%100!=0) || (year%400==0)) // 方法 1
    }
}

```

```

        System.out.println(year+" 年是闰年");
    else
        System.out.println(year+" 年不是闰年");

    year=2000; // 方法 2
    if (year%4!=0)
        leap=false;
    else if (year%100!=0)
        leap=true;
    else if (year%400!=0)
        leap=false;
    else
        leap=true;

    if (leap==true)
        System.out.println(year+" 年是闰年");
    else
        System.out.println(year+" 年不是闰年");

    year=2050; // 方法 3
    if (year%4==0) {
        if (year%100==0) {
            if (year%400==0)
                leap=true;
            else
                leap=false;
        }
        else
            leap=false;
    }
    else
        leap=false;

    if (leap==true)
        System.out.println(year+" 年是闰年");
    else
        System.out.println(year+" 年不是闰年");
}
}

```

程序说明：方法 1 用一个逻辑表达式包含了所有的闰年条件；方法 2 使用了 if...else 语句的复合形式；方法 3 则通过大括号“{}”对 if...else 进行匹配来实现闰年的判断。大家可以根据程序对比这三种方法，体会其中的联系和区别，在不同的场合选用适当的方法。

### 3.2.4 switch 开关语句

虽然嵌套的条件语句可实现多个分支处理,但嵌套太多时容易出错和混乱,这时可以使用开关语句 switch 处理。实际上开关语句 switch 也是一种 if... else 结构,不过它使你在编程时很容易写出判断条件,特别是有很多条件选项的时候。

开关语句 switch 的语法结构如下:

```
switch(表达式){  
    case 常量 1:  
        语句 1;  
        break;  
    case 常量 2:  
        语句 2;  
        break;  
    .....  
    default:  
        语句 n;  
}
```

其中 switch、case、default 是关键字,default 子句可以省略。开关语句先计算表达式,然后将表达式值与各个常量比较,如果表达式值与某个常量相等,就执行该常量后面的语句。如果都不相等,就执行 default 下面的语句。如果无 default 子句,就什么都不执行,直接跳出开关语句。

使用开关语句时,注意以下两个问题:

- (1) case 后面的常量必须是整数或字符型,而且不能有相同的值;
- (2) 通常在每一个 case 中都应使用 break 语句提供一个出口,使流程跳出开关语句。否则,在第一个满足条件 case 后面的所有语句都会被执行,这种情况叫做落空。看下面分别加上和去掉 break 语句的例子。

例 3.5 switch 语句的使用,有 break 语句。输出结果如图 3.3 所示。

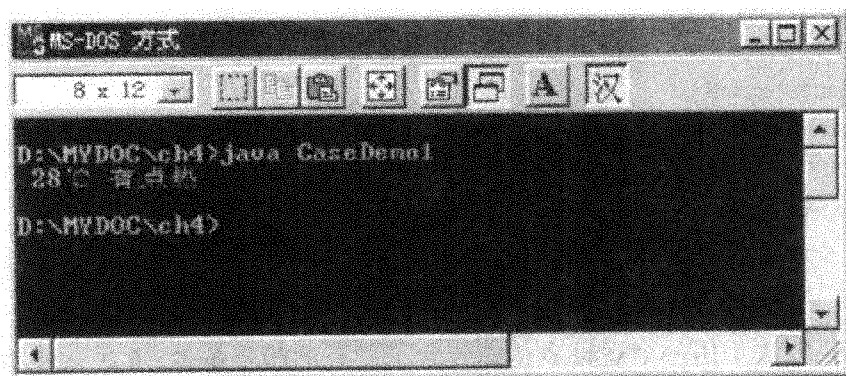


图 3.3

```
public class CaseDemo1 {
```

```

public static void main(String args[]) {
    int c=28;
    switch (c<10? 1:c<25? 2:c<35? 3:4) {
        case 1;
            System.out.println(" "+c+"℃ 有点冷");
            break;
        case 2;
            System.out.println(" "+c+"℃ 正合适");
            break;
        case 3;
            System.out.println(" "+c+"℃ 有点热");
            break;
        default;
            System.out.println(" "+c+"℃ 太热了");
    }
}
}

```

**例 3.6** switch 语句的使用,无 break 语句。输出结果如图 3.4 所示。

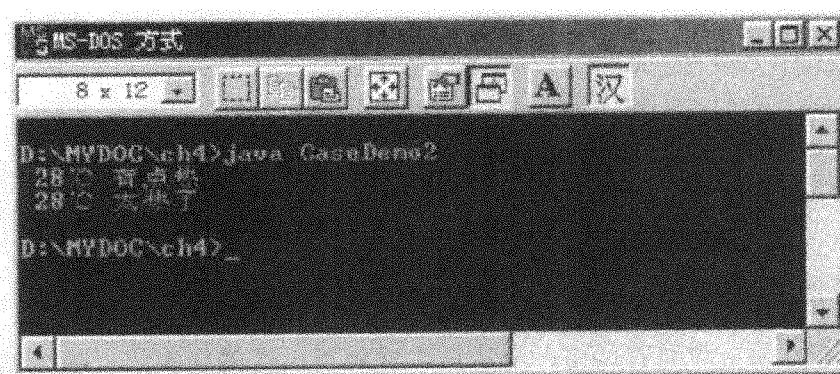


图 3.4

```

public class CaseDemo2 {
    public static void main(String args[]) {
        int c=28;

        switch (c<10? 1:c<25? 2:c<35? 3:4) {
            case 1; System.out.println(" "+c+"℃ 有点冷");
            case 2; System.out.println(" "+c+"℃ 正合适");
            case 3; System.out.println(" "+c+"℃ 有点热");
            default; System.out.println(" "+c+"℃ 太热了");
        }
    }
}

```

**程序说明:**从这两个画面上可以看出 break 语句的作用。例 3.6 由于缺少 break 语

句,使得程序执行完 case 3 下面的语句,紧接着又执行了 default 下面的语句。

这两个程序采用了转换方法,将判断条件的取值最终转换为数值,请读者自行分析三元运算符的作用。

### 3.3 循环语句

到目前为止,我们看到的都是线性的程序,即每行命令只有一次执行的机会(选择语句则会忽略若干行)。能否重复执行一些语句呢?利用循环语句就可以实现这个目的。循环可使程序根据一定的条件重复执行某一部分程序代码,直到满足终止循环条件为止。

Java 中提供的循环语句有:确定次数循环(for)、条件循环(while)、先执行后判定循环(do)。

#### 3.3.1 for 循环语句

如果希望程序的一部分内容按固定的次数重复执行,通常可以使用 for 循环。for 循环采用一个计数器控制循环次数,每循环一次计数器就加 1,直到完成给定的循环次数为止。

下面的一段程序利用 for 循环语句为数组赋值。

```
int a[]=new int[10];
for (int i=0; i<10; i++) {
    a[i]=0;
}
```

这段程序把整型数组 a 中的所有元素都赋值为 0。

**例 3.7** 按 5 度的增量打印出一个从摄氏度到华氏度的转换表。输出结果如图 3.5 所示。

```
class CtoF {
    public static void main (String args[]) {
        int fahr,cels;
        System.out.println("摄氏度 华氏度");

        for (cels=0; cels<=40; cels+=5) {
            fahr=cels*9/5+32;
            System.out.println(" "+cels+" "+fahr);
        }
    }
}
```

从上面的例子中我们归纳出 for 循环的语法结构为:

```
for (表达式 1; 表达式 2; 表达式 3)
    循环体
```

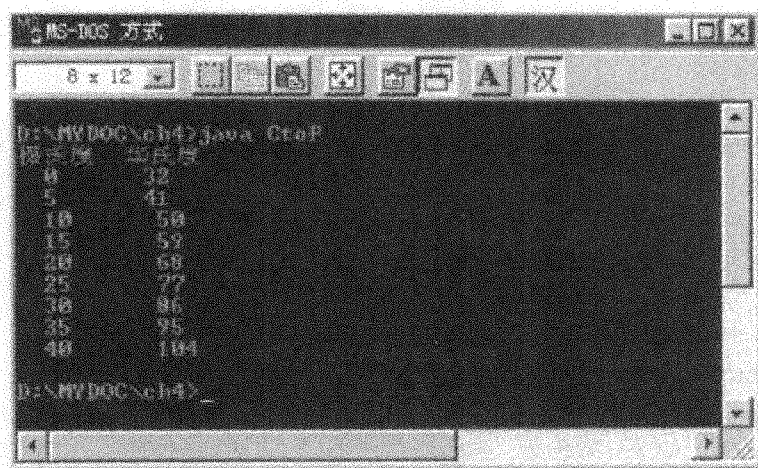


图 3.5

其中表达式 1 指出计数器的初值,是一个赋值语句;表达式 2 指出循环结束条件,是一个逻辑表达式;表达式 3 指出计数器每次的增量,是一个赋值语句。

注意:计数器可在 for 语句之前定义,也可在循环括号中定义。计数器增量为 1 时常写成增量运算的形式,以加快运算速度。根据需要,增量可以大于 1。增量计算也可以放在循环体中进行,即把表达式 3 移到循环体内的适当位置,原位置为空。

使用 for 循环,还要注意初值、终值和增量的搭配。终值大于初值时,增量应为正值,终值小于初值时,增量应为负值。编程时必须密切关注计数器的改变,这是实现正常循环避免陷入死循环的关键。

### 3.3.2 while 循环语句

while 循环不像 for 循环那么复杂,while 循环只需要一个条件判断语句,便可以进行循环操作。

**例 3.8** 下面这个程序可接受从终端输入的数字,并显示得到的奖品。其中使用了开关语句 Switch 和 while 循环语句。输出结果如图 3.6 所示。

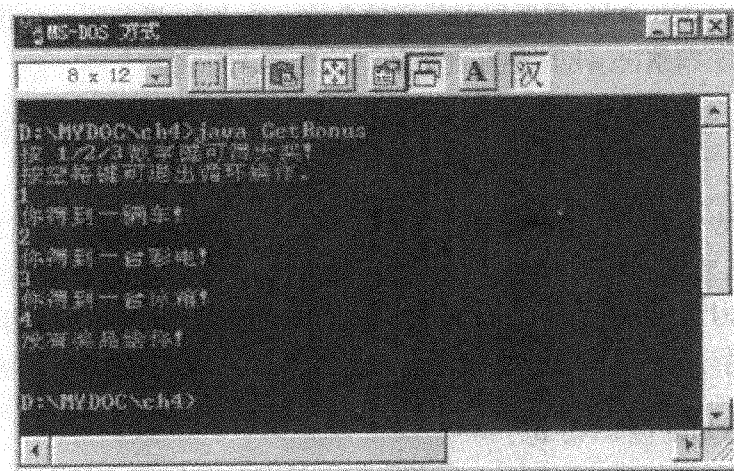


图 3.6

```

import java.io. * ;

class GetBonus {
    public static void main(String args[]) throws IOException {
        char ch;
        System.out.println("按 1/2/3 数字键可得大奖!");
        System.out.println("按空格键可退出循环操作。");

        while ((ch=(char)System.in.read()) != ' ') {
            System.in.skip(2); // 跳过回车键
            switch (ch) {
                case '1':
                    System.out.println("你得到一辆车!");
                    break;
                case '2':
                    System.out.println("你得到一台彩电!");
                    break;
                case '3':
                    System.out.println("你得到一台冰箱!");
                    break;
                default:
                    System.out.println("没有奖品给你!");
            }
        }
    }
}

```

程序说明:while 循环从键盘读入字符直至输入一个空格退出。这里使用了系统的输入方法 System.in.read() 和 throws IOException 抛出异常,后面的章节会专门介绍。循环内部的第一条语句处理回车键,将每次输入时按下的回车键屏蔽掉。switch 语句先把从键盘读入字符和给定字符比较,当发现相等时,就显示得到的奖品名称。然后经由 break 语句跳出 switch 语句,程序回到等待键盘输入的状态,进行下一轮循环。

从上面的例子归纳出 while 循环的语法结构为如下形式:

```

while (条件表达式)
    循环体

```

其中 while 是关键字。每次循环之前都要计算条件表达式,其值为真时,就执行一次循环体中的语句,然后再计算条件表达式,决定是否再次执行循环体中的语句;如果条件表达式的值为假时,就跳出循环体,执行循环体下面的语句。while 循环中的条件表达式是逻辑表达式,循环体中一定要有改变条件表达式值的语句,否则会陷入死循环。

### 3.3.3 do...while 循环语句

do...while 循环与 while 循环相反,是先执行循环体中的语句,再计算 while 后面的



条件表达式,若条件表达式值为假则跳出循环,否则继续下一轮循环。

什么时候使用 do... while 循环呢? 有些情况下,不管条件表达式的值是为真还是假,你都希望把指定的语句至少执行一次,那么就应使用 do... while 循环,看下面的例子。

**例 3.9** 求前 n 个数之和,设  $n = 10$ ,输出结果如图 3.7 所示。

```
class Sum {
    public static void main(String args[]) {
        int n=1;
        int sum=0;

        do
            sum+=n++;
        while (n<=10);
        System.out.println("前"+n+"个数的总和为:"+sum);
    }
}
```

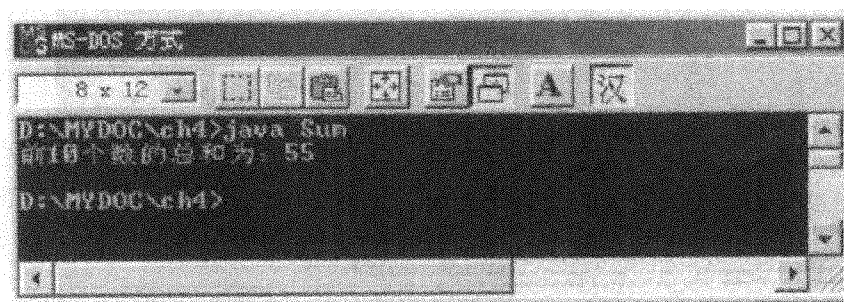


图 3.7

我们归纳出 do... while 循环的语法结构为:

```
do {
    循环体
} while (条件表达式);
```

其中 do、while 是关键字。程序首先执行 do 下面的循环体,然后计算 while 后面条件表达式的值,如果其值为 true,则重复执行循环体,否则,就结束循环。

与 while 循环相同,do 循环在循环体中也一定要有改变条件表达式值的语句,否则会陷入死循环。do... while 循环控制并不是很常用,但有时却非常重要,使用时特别注意不要忘记了 while 语句结尾处的分号“;”。

### 3.3.4 循环语句的嵌套

在实际使用中,上面介绍的循环语句可嵌套使用,请看下面的例子。

**例 3.10** 分别用 while、do 和 for 循环语句实现累计求和。输出结果如图 3.8 所示。

```

public class Sum1 {
    public static void main(String args[]) {
        int n=10, sum=0;
        while (n>0) {
            sum=0;
            for (int i=1; i<=n; i++)
                sum+=i;
            System.out.println("前"+n+"个数的总和为:"+sum);
            n--;
        }
    }
}

```

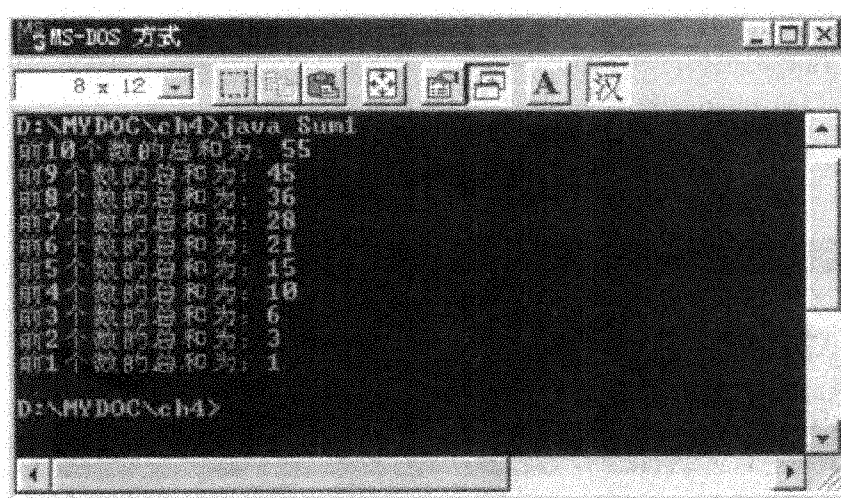


图 3.8

### 3.3.5 循环语句小结

一个循环一般应包括以下 4 部分内容。

- (1) 初始化部分 用来设置循环的一些初始条件,计数器清零等;
- (2) 循环体部分 这是反复被执行的一段代码,可以是单语句,也可以是复合语句;
- (3) 迭代部分 这是在当前循环结束,下一次循环开始时执行的语句,常用来使计数器加 1 或减 1;
- (4) 终止部分 通常是一个布尔表达式,每一次循环要对该表达式求值,以验证是否满足循环终止条件。

## 3.4 跳转语句

跳转语句可以无条件改变程序的执行顺序。Java 支持三种跳转语句: break、continue 和 return。

### 3.4.1 break 语句

break 语句提供了一种方便的跳出循环的方法。使用 break 语句可以立即终止循环，即使循环没有结束也如此。

例 3.11 break 语句的使用。输出结果如图 3.9 所示。

```
class BreakDemo {  
    public static void main(String args[]) {  
        boolean test=true;  
        int i=0;  
        while (test) {  
            i=i+2;  
            System.out.println("i="+i);  
            if (i>=10)  
                break;  
        }  
        System.out.println("i 为"+i+"时循环结束");  
    }  
}
```

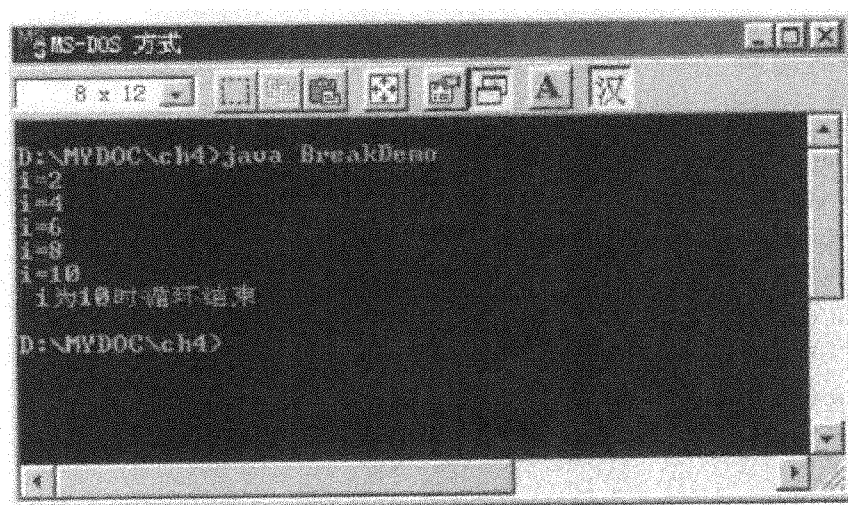


图 3.9

执行这个程序时，尽管 while 条件表达式始终为真，但实际上只循环了 5 次。这是因为当 i 大于等于 10 时遇到了 break 语句，使流程跳出了循环。

### 3.4.2 带标号的 break 语句

标号是标记程序位置的标识符。break 语句只能跳转到结构下面的第一条语句上，而带标号的 break 语句可直接跳转到标号处。例如，当你陷在多重循环中又想退出循环时会怎样呢？正常的 break 只退出一重循环，如果要退出多重循环，可以使用带标号的 break 语句。它的语法结构如下：

标识符:

...

break 标识符;

例 3.12 带标号的 break 语句的使用。输出结果如图 3.10 所示。

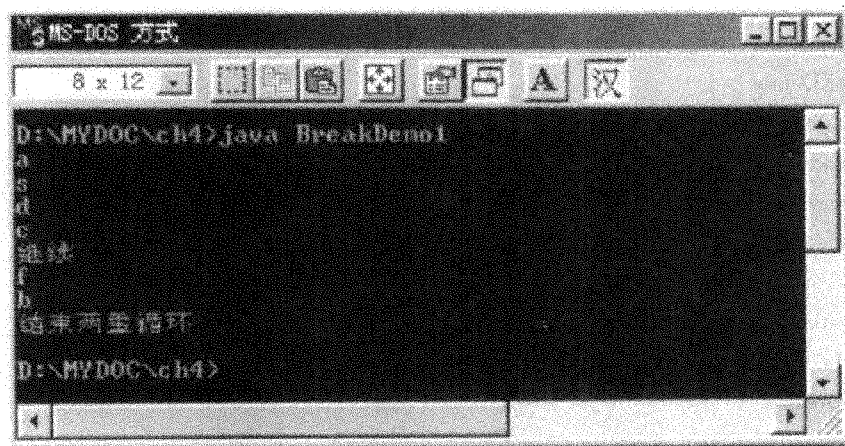


图 3.10

```
class BreakDemo1 {
    public static void main(String args[]) throws java.io.IOException {
        char ch;

        lab1: // 此处为标号
        for (int i=0; i<4; i++)
            for (int j=0; j<4; j++) {
                ch=(char)System.in.read();
                System.in.skip(2);
                if (ch=='b')
                    break lab1;
                if (ch=='c')
                    System.out.println("继续");
            }
        System.out.println("结束两重循环");
    }
}
```

在这个例子中,正常情况下程序可从键盘接受 16 个输入字符,当输入“b”时,break lab1 语句就会结束二重循环。

### 3.4.3 continue 语句

continue 语句只能用在循环结构中,它跳过循环体中尚未执行的语句,重新开始下一轮循环。下面通过一个例子来说明它的使用方法。

例 3.13 下面的程序可以输出 1~9 中除 6 以外所有偶数的平方值。输出结果如图

3.11 所示。



图 3.11

```
class ContDemo {
    public static void main(String args[]) {
        for (int i=2;i<=9;i+=2) {
            if (i==6)
                continue;
            System.out.println(i+" 的平方 = "+i*i);
        }
    }
}
```

#### 3.4.4 带标号的 continue 语句

Java 也支持带标号的 continue 语句,它通常用在嵌套循环的内循环中,你可以用标号标出你想跳到哪一条语句继续重复执行程序。它的语法结构如下:

标识符:

...

continue 标识符;

**例 3.14** 带标号的 continue 语句的使用。输出结果如图 3.12 所示。

```
class ContDemo1 {
    public static void main(String args[]) {
        lab1:
        for (int i=1; i<=3; i++)
            for (int j=1; j<=3; j++) {
                System.out.println("i="+i+" j="+j);
                if (i+j>3) {
                    System.out.println("Continue");
                    continue lab1;
                }
                System.out.println("i="+i+" j="+j);
            }
    }
}
```

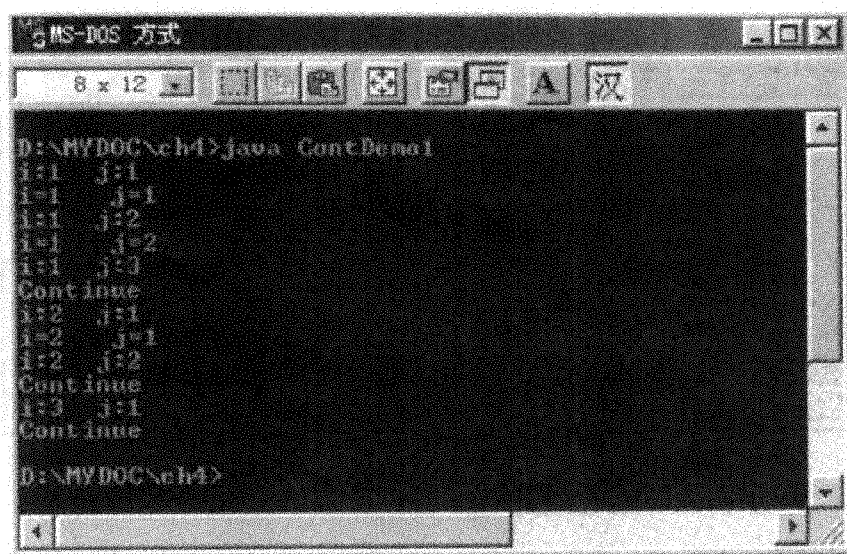


图 3.12

例 3.15 求 100~200 间的所有素数。该例通过一个嵌套的 for 循环来实现。输出结果如图 3.13 所示。

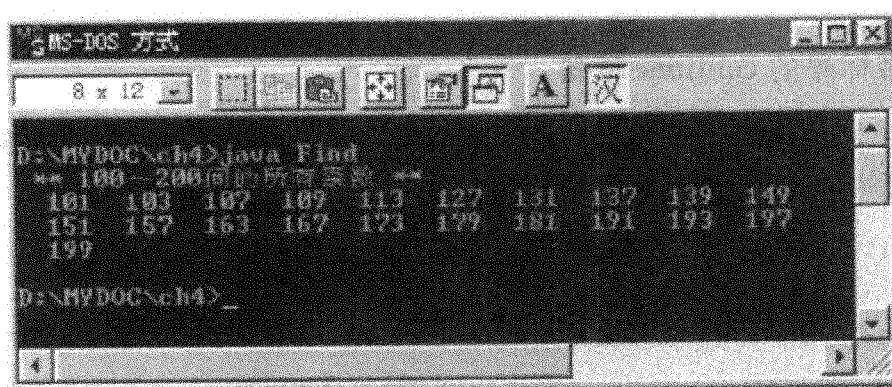


图 3.13

```
public class Find {
    public static void main(String args[]) {
        System.out.println(" * * 100~200 间的所有素数 * *");
        int n=0;

        outer:
        for (int i=101; i<200; i+=2) {
            int k=15;
            for (int j=2; j<=k; j++)
                if (i%j==0)
```

```

        continue outer;
    System.out.print(" " + i);
    n++;
    if (n < 10)
        continue;
    System.out.println();
    n = 0;
}
System.out.println();
}
}

```

注意:程序中分别使用了 continue 语句和带标号的 continue 语句。在输出方法上分别使用了系统的 println 方法和 print 方法,二者的区别在于前者输出时换行,后者输出时不换行。

### 3.4.5 return 语句

return 语句用于方法的返回上,当程序执行到 return 语句时,终止当前方法的执行,返回到调用这个方法的语句。return 语句通常位于一个方法体的最后一行,有带参数和不带参数两种形式,带参数形式的 return 语句退出该方法并返回一个值。

当方法用 void 声明时,说明不需要返回值(即返回类型为空),应使用不带参数 return 语句。不带参数的 return 语句也可以省略,当程序执行到这个方法的最后一条语句时,遇到方法的结束标志“}”就自动返回到调用这个方法的程序中。

带参数的 return 语句有如下语法形式:

```
return 表达式;
```

当程序执行到这个语句时,首先计算“表达式”的值,然后将表达式的值返回到调用该方法的语句。返回值的数据类型必须与方法声明中的返回值类型一致,可以使用强制类型转换来使类型一致。

例 3.16 带参数的 return 语句的使用。输出结果如图 3.14 所示。



图 3.14

```

class CircleArea {
    final static double PI=3.14159;

    public static void main(String args[]) {
        double r1=8.0, r2=5.0;
        System.out.println("半径为"+r1+"的圆面积="+area(r1));
        System.out.println("半径为"+r2+"的圆面积="+area(r2));
    }

    static double area(double r) {
        return (PI * r * r);
    }
}

```

程序中定义了双精度数据类型的 area 方法,相应地,return 语句带有一个双精度数据类型的返回值。在 main 方法中,引用了这个方法,输出了圆面积的数值。area 方法的功能就像函数一样,你填写一个数值,它为你提供一个返回值。

读者可能会注意到 Java 语言没有 goto 语句,这对于那些惯于使用 goto 语句的程序员来讲可能会有些不适应。但有经验的程序员一定会明白,goto 语句实际上是导致程序结构混乱的根源之一,使你日后要花大量时间去找出一个程序是干什么的,它最终所引起的问题比它可能带来的好处要大得多。正由于这些原因,Java 的设计者果断地删去了它。

## 习 题

- 3-1 编写程序,比较两个数的大小,并按从大到小的次序输出。
- 3-2 编写程序,从 10 个数中找出最大值。
- 3-3 编写程序,根据考试成绩的等级打印出百分制分数段。设 A 为 90 分以上、B 为 80 分以上、C 为 70 分以上、D 为 60 分以上、E 为 59 分以下。要求在程序中使用开关语句。
- 3-4 编写程序,计算  $n$  的阶乘( $n!$ ),设  $n=10$ 。
- 3-5 编写程序,计算数学常数  $e$  的值, $e=1+1/1!+1/2!+1/3!+\dots$ 。
- 3-6 编写程序,输出 1~100 间的所有奇数。
- 3-7 编写程序,输出以下数据:

N	10 * N	100 * N	1000 * N
1	10	100	1000
2	20	200	2000
3	30	300	3000
4	40	400	4000
5	50	500	5000



**3-8** 回文是一种从前向后读和从后向前读都一样的文字或数字,如 12121、66、335533,编写程序,利用随机数生成一个 5 位数,并判断它是不是回文。

**3-9** 本章例 3.1 的输出结果是什么?

**3-10** 以下程序的输出结果是什么?

```
public class Test1 {  
    public static void main(String args[]) {  
        int y, x=1, total=0;  
        while(x<=10) {  
            y=x * x;  
            System.out.println(y);  
            total+=y;  
            ++x;  
        }  
        System.out.println("total is "+total);  
    }  
}
```

**3-11** 以下程序的输出结果是什么?

```
public class Test2 {  
    public static void main(String args[]) {  
        int count=1;  
        while(count<=10) {  
            System.out.println(count%2==1?"* * * * ":"++++++");  
            ++count;  
        }  
    }  
}
```

**3-12** 完整下面的程序,利用 break 语句和带标号的 break 语句分别退出一重循环和二重循环。

```
for (i=0; i<10; i++)  
{  
    int j=i * 10  
    while(j<100)  
    {  
        if (j==10)  
            break;  
        j=j+5;  
    }  
}
```



# 第4章

## 面向对象编程

在前几章里,大部分的例子都没有过多的涉及 OOP(面向对象编程)方面的内容,这是为读者提供的一种过渡。现在你已经掌握了 Java 语言的基本内容,正在跃跃欲试,要编写一个体现 OOP 风格的 Java 程序。从本章起,我们将为你提供这样的机会。随着学习内容的深入,你的 Java 编程能力将逐渐增强。

### 4.1 类

Java 程序由类组成,一个程序至少要包含一个类,Java 程序员的任务就是设计出这样的类来实际问题。创建类时既可以从父类继承,也可以自行定义,我们先通过几个例子来说明如何创建类。

#### 4.1.1 类的创建

例 4.1 下面这个程序将在屏幕上输出两个矩形,并伴有文字输出,见图 4.1。

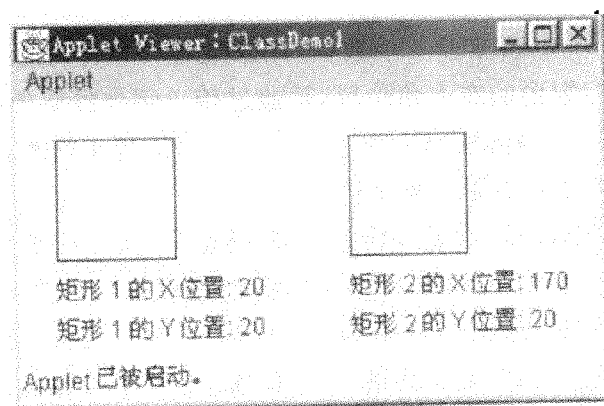


图 4.1

```
import java.awt.*;  
import java.applet.Applet;
```

```

public class ClassDemol extends Applet {
    private int x, y, width, height;

    public void init() {
        width=60;
        height=60;
    }

    public void setPosition(int xPos, int yPos) {
        x=xPos;
        y=yPos;
    }

    public void paint(Graphics g) {
        setPosition(20,20);
        g.drawRect(x, y, width, height);
        g.drawString("矩形 1 的 X 位置: "+x, 20,100);
        g.drawString("矩形 1 的 Y 位置: "+y, 20,120);

        setPosition(170,20);
        g.drawRect(x, y, width, height);
        g.drawString("矩形 2 的 X 位置: "+x, 170,100);
        g.drawString("矩形 2 的 Y 位置: "+y, 170,120);
    }
}

```

程序中的 ClassDemol 是一个子类,继承自系统类 Applet。一般来说,用继承的方式创建一个类是最常用也是最简单的方法,子类除了可以直接使用父类的数据和方法外,还可以有自己的数据和方法。

我们在 ClassDemol 中添加了 4 个数据,用来确定一个矩形的 4 个参数。新增的方法 setPosition 用于改变矩形的左上角位置,是通过传递两个整型参数实现的。

init 和 paint 方法属于 Applet,ClassDemol 对它们进行了修改,用 init 方法为矩形的宽和高赋值,用 paint 方法画出矩形和字符串。

在 paint 方法中,先调用 setPosition 方法设定矩形的位置,然后用 Graphics 的 drawRect 画出矩形。drawRect 需要 4 个参数,我们将代表矩形的 4 个数据 x、y、width 和 height 传递给它就可画出矩形。

drawString 需要 3 个参数:要输出的字符串、输出位置 x 和 y。你可能会注意到,字符串参数由一个字符串常量和一个整型数相加而成,这在 Java 中是合法的,因为 Java 中的字符串是一个 String 类,有很多特性。那么 g 是从何而来的呢? g 是一个图形对象,由 Applet 启动时传递过来。

最后要指出的是,上面的例子完全可以采用更简单的方法来编程,我们这样做的目的主要是为了说明如何创建一个类。

**例 4.2** 下面这个程序是改写后的例 4.1, 输出结果是一样的, 见图 4.1。

```
import java.awt.* ;
import java.applet.Applet;

public class ClassDemo2 extends Applet {
    MyBox b1=new MyBox();
    MyBox b2=new MyBox(170,20,60,60);

    public void paint(Graphics g) {
        b1.setPosition(20,20);
        b1.setSize(60,60);
        b1.draw(g);
        g.drawString("矩形 1 的 X 位置: "+b1.getX(), 20, 100);
        g.drawString("矩形 1 的 Y 位置: "+b1.getY(), 20, 120);

        b2.draw(g);
        g.drawString("矩形 2 的 X 位置: "+b2.getX(), b2.getX(), b2.getY()+80);
        g.drawString("矩形 2 的 Y 位置: "+b2.getY(), b2.getX(), b2.getY()+100);
    }
}

class MyBox {
    private int x, y, width, height;

    MyBox() {
        x=0;
        y=0;
        width=0;
        height=0;
    }

    MyBox(int xPos, int yPos, int w, int h) {
        x=xPos;
        y=yPos;
        width=w;
        height=h;
    }

    public void setPosition (int xPos, int yPos) {
        x=xPos;
        y=yPos;
    }
}
```

```

public void setSize (int w, int h) {
    width=w;
    height=h;
}

public int getX() {
    return x;
}

public int getY() {
    return y;
}

public void draw(Graphics g) {
    g.drawRect(x, y, width, height);
}
}

```

例 4.2 看起来更具 OOP 风格。ClassDemo2 和例 4.1 的 ClassDemo1 同属一类,但做的事情更少。ClassDemo2 使用 new 操作符创建了两个对象实例 b1 和 b2,然后在 paint 方法中向它们传递信息,操纵这两个对象完成给定的任务。

MyBox 是自定义类,虽然没有明确指出其父类,但仍存在继承关系,它的父类就是 Java 的根类 Object。每一种 OOP 语言都有一个这样的根类,然后衍生出该语言的系统类。自定义类需要自行设计类的状态和行为,即确定它的成员变量和方法。我们设计的 ClassDemo2 有 4 个私有数据,2 个构造方法和 5 个成员方法。

4 个数据声明为私有数据,可以禁止外来访问,起到保护的目的。方法都声明为公共的方法,以接收外部信息,改变类的行为。

MyBox() 和 MyBox(int, int, int, int) 同为类 MyBox 的构造方法,方法名相同而参数不同,这是类的多态性的体现。程序可以根据参数的不同,自动调用正确的构造方法。

由 MyBox 创建的两个实例对象 b1 和 b2 具有 MyBox 的全部特征,在 ClassDemo2 中不能直接访问 b1 和 b2 的成员变量,它们的成员变量只能通过自身的方法来访问。通过 setPosition 设定 MyBox 的位置,通过 setSize 改变 MyBox 的大小,通过 getX 和 getY 获取 MyBox 的位置。

上面两个程序的设计,完整体现了 OOP 的基本思想:抽象、封装、继承、多态。

在软件工程中,常常将具体事物与主题无关的部分剔除,而将注意力集中在与主题相关的事物特征上,得出一个抽象的软件模型。OOP 采用数据抽象的方法构造软件模型,将事物的各种特征抽象为不同的数据类型,它们体现了事物的属性、功能和作用。这些数据类型的集合就构成了 OOP 的对象。

如例 4.2,画一个矩形时,我们最关心的是它的位置和大小,以及围绕这些数据所实施的操作。将表示矩形位置和大小 4 个数据以及围绕它们实施的存取、画出等方法结合在一起,就抽象成为我们所关心的对象。这里,我们用一个抽象概念——类来表示。

封装是抽象的具体实现。封装就是用操作方法把数据封闭到对象中,形成以数据为核心,以方法为外壳的对象。封装能保护对象的数据免受外界的更改,消除了由此带来的对程序的不可知影响。封装的结果是形成了独立的和完整的程序模块,它们之间通过被授权的操作方法来传递消息,达到改变对象状态的目的,这是提高程序健壮性的有力保证。

关于类的另外两个基本概念:继承和多态,我们将在下一章展开讨论。

#### 4.1.2 类的修饰

类通过关键字 `class` 来创建,下面的声明语句格式给出了可能出现的符号和顺序:

```
[public] [abstract] [final] class 类名 [extends 父类] [implements 接口] {  
    类体  
}
```

`class` 前面的关键字称为类的修饰符。

##### 1. `class` 类名

`class` 关键字告诉编译器这是一个类,类名必须是合法的标识符。

##### 2. `public` 公共类

在没有任何修饰符的默认情况下,类只能被同一个源程序文件或同一个包中的其他类使用,如例 4.2 中的 `MyBox` 只能被 `ClassDemo2` 使用。加上 `public` 修饰符后,类可以被任何包中的类使用,有关包的概念请参考后面章节的内容(目前可把包当作一个源程序文件)。

注意:在同一个源程序文件中不能出现两个以上的 `public` 类,否则编译器会告诉你将第二个 `public` 类放在另一个文件中。

##### 3. `abstract` 抽象类

有时,你定义的代表了一个抽象的概念,就不能用它来实例化一个对象。例如,现实世界中食品这个概念,你不可能有一个食品对象,你见到的是一些诸如饼干、苹果、巧克力等具体对象。食品代表着一个抽象概念:能吃的东西。饼干、苹果、巧克力等是食品的子类,更加具体一些,它们才可以产生对象,如某某牌子的甜饼干、某某牌子的咸饼干。没有见到过某某牌子的甜食品、某某牌子的咸食品。

同样在 OOP 中,你可能建立了一个不需要产生对象的类。例如, `java.lang` 包中的 `Number` 类代表了数这个抽象概念,可以用它在程序中产生一个数的子类,如 `Integer` 或 `Float`,但从 `Number` 中直接生成对象是没有意义的。

`abstract` 说明的类称为抽象类,不能用它实例化一个对象,它只能被继承。

##### 4. `final` 最终类

一个最终类不可能有子类,也就是说它不能被继承。为什么要把一个类说明为最终

类呢？有两点理由：为了提高系统的安全性和出于对一个完美类的偏爱。

黑客常用的一个攻击技术是设计一个子类，然后用它替换原来的父类。子类和父类很相像，但做的事情却大不一样。为防止这样的事情发生，你可以把你的类声明为最终类，不让黑客有机可乘。

Java 中的 String 类设计为一个最终类就是出于这样的目的，一个方法或对象无论何时使用 String，Java 解释器总是使用系统的 String 类而不是其他，这样就保证任何字符串不含有奇怪的、不希望的或不可预料特性。此外，程序员有时也会把一些无懈可击的类声明成最终类，不让别人把它们改得面目全非。

final 和 abstract 不能同时修饰一个类，这样的类是没有意义的。

## 5. extends 父类

说明类是从父类继承下来的子类，父类必须有定义。

从父类继承，可以提高代码的重用性，不必从头开始设计程序。大部分情况下应该利用继承的手段编程，只有在没有合适的类可以继承时才自己设计类。

## 6. implements 接口

说明类实现的接口，接口必须有定义。

接口是消息传递的通道，通过接口，消息才能传递到处理方法中进行处理。implements 说明你的类可以实现的一个或多个接口，如果有多个接口，要用逗号分隔。关于接口的设计和使用请参考后面章节的内容。

### 4.1.3 类体

类体包含在一对大括号内，是所有代码的集合；这些代码将在实例对象的生命周期内起作用。自定义类应提供构造方法，它和类同名，完成对象初始化工作。变量表明类和类创建的对象的状态。方法用于实现类和类创建的对象的行为。在少数情况下，类还要提供一个 finalize 方法。当对象使用完毕后，由 finalize 方法清除对象。

在类体中定义的变量和方法都称为类的成员，称为成员变量和成员方法。成员变量可以是任何数据类型，也可以是另外一个对象。如例 4.2 中，ClassDemo2 是一个子类，不需要构造方法。但它的两个成员变量是由 MyBox 类创建的对象。

### 4.1.4 类的构造方法

自定义类实际上是 Java 根类 Object 的子类。Object 仅提供了一些关于类本身的方法，很少具体功能。因此，自定义类必须添加状态和行为代码，使类具有某种功能。

构造方法对于类是十分重要的，new 操作符为对象分配内存后将调用类的构造方法确定对象的初始状态。

对于简单的自定义类，可以不设计构造方法，在声明变量的同时进行赋值，那么创建对象时，将使用默认构造方法。默认的构造方法没有参数，属于父类，如果父类中没有提供构造方法，将产生编译错误。如果初始化时还要执行一些其他命令，就必须设计构造方



法,将这些命令放在构造方法中,因为 Java 规定命令语句不能出现在类体中,只能放在方法中。

构造方法的名称和类同名,没有返回类型。尽管构造方法看起来和一般的成员方法没有差别,但它不是方法,也不是类的成员。因此,构造方法不能由程序员直接调用,只能由 new 操作符调用。

在同一个类中可以定义多个构造方法,名字相同而参数可以不同。它们是以参数的个数来区分的,这种情况称为方法重载(overload)。创建对象时,程序员可根据需要选择合适的构造方法。

如例 4.2 中定义的两个构造方法:

```
MyBox() {  
    x=0;  
    y=0;  
    width=0;  
    height=0;  
}  
  
MyBox(int xPos, int yPos, int Width, int Height) {  
    x=xPos;  
    y=yPos;  
    width=Width;  
    height=Height;  
}
```

第一个是默认的构造方法 MyBox(),没有任何参数,它将把 MyBox 的 4 个成员变量赋值为 0。

第二个构造方法 MyBox(int, int, int, int) 有 4 个参数,创建对象时可以传递 4 个参数作为 MyBox 的初值。不同的构造方法可以为程序员提供更加灵活的选择。

## 4.2 成员变量

成员变量描述了类和对象的状态,有时也称为属性、数据、域(field)。对成员变量的操作实际上就是改变对象的状态,使之能满足程序的需要。与类相似,成员变量也有很多修饰符,用来控制对成员变量的访问。下面,我们对成员变量和修饰符作进一步讨论。

### 4.2.1 成员变量的声明

成员变量的声明必须放在类体中,通常是在成员方法之前。在方法中声明的变量不是成员变量,而是方法的局部变量,二者是有区别的。

例 4.3 显示当前日期和时间,运行结果见图 4.2。

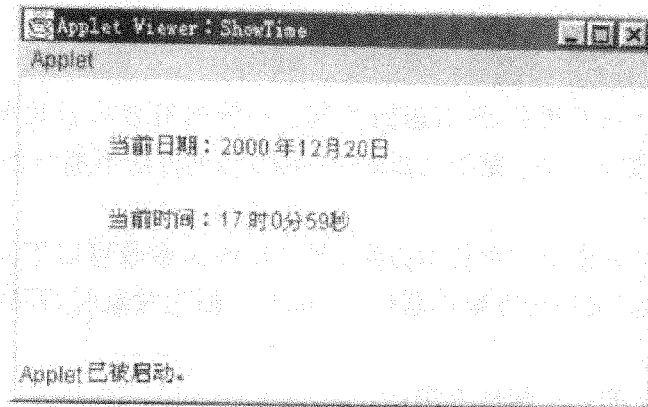


图 4.2

```
import java.awt.Graphics;
import java.applet.Applet;
import java.util.Calendar;

class Time {
    private Calendar t;
    private int y, m, d, hh, mm, ss;

    Time {
        t = Calendar.getInstance();
        y = t.get(t.YEAR);
        m = t.get(t.MONTH) + 1;
        d = t.get(t.DATE);
        hh = t.get(t.HOUR_OF_DAY);
        mm = t.get(t.MINUTE);
        ss = t.get(t.SECOND);
    }

    public String getDate() {
        return y + " 年" + m + " 月" + d + " 日";
    }

    public String getTime() {
        String s = hh + " 时" + mm + " 分" + ss + " 秒";
        return s;
    }
}

public class ShowTime extends Applet {
    Time t = new Time();

    public void paint(Graphics g) {
```

```

        g.drawString("当前日期:" + t.getDate(), 50, 40);
        g.drawString("当前时间:" + t.getTime(), 50, 80);
    }
}

```

Time 是一个自定义类,它可以返回系统当前日期和时间。Java 中获取日期和时间的方法有很多。我们采用了 Calendar 类中的方法,它有一个 getInstance 方法,可获取系统当前日期和时间,并保存到 Calendar 对象的相应成员变量中。通过 get 方法分别取出日期和时间分量,再赋值给类的成员变量。最后将这些成员变量分别合成为日期字符串和时间字符串,由成员方法 getDate 和 getTime 返回。

主类 ShowTime 仅仅声明了一个 Time 对象,然后调用对象的方法取得日期和时间并输出到窗口的指定位置。请注意 Time 成员变量的声明,它们用 private 修饰,并在声明的同时进行赋值,这种情况下就不需要构造方法了。成员变量声明时要注意顺序,一个重要的原则是:使用变量之前必须先声明它。

我们在 getTime 方法中特意声明了一个字符串变量 s,用来说明局部变量和成员变量的区别。s 仅在 getTime 中有定义,成员变量在整个类中有定义。还有一个很重要的区别是,成员变量有默认值,局部变量没有默认值。这就意味着即使不对成员变量赋初值,编译器也会按数据类型为每一个成员变量赋默认值,使对象有一个确定的状态。而局部变量必须人工赋初值,否则编译器会给出变量未赋值的警告。

## 4.2.2 成员变量的修饰

以上几个例子的成员变量声明都是比较简单的,只出现了一个访问控制修饰符 private。实际上,成员变量的声明可以是非常复杂的。

成员变量的声明语句有如下格式:

```

[public] [private] [protected] [package] // 访问控制修饰符
[static] [final] [transient] [volatile]
类型 名称

```

### 1. 访问控制

表 4.1 给出了访问控制修饰符的作用范围。

表 4.1 修饰符的作用范围

修饰符	类	子类	包	所有类和包
public	✓	✓	✓	✓
private	✓			
protected	✓	✓ *	✓	
package	✓		✓	

(1) public 公共变量可被任何包中的任何类访问,只有在确认任何外部访问都不会

带来不良后果的情况下才将成员声明为公共的。公共变量对任何类都是可见的,因此它没有秘密可言,不具有数据保护功能。例如下面这两段程序;

```
package P1;
public class ClassA {
    public int isPublic;
    public void publicMethod() {
        System.out.println("publicMethod");
    }
}
```

```
package P2;
import P1. * ;
class ClassB {
    void accessMethod() {
        ClassA a=new classA();
        a.isPublic=10;
        a.publicMethod();
    }
}
```

(2) private 私有变量只能被声明它的类所使用,拒绝任何外部类的访问。私有变量是不公开的,它们得到了最好的保护。例如:

```
class ClassA {
    private int isPrivate;
    private void privateMethod() {
        System.out.println("privateMethod");
    }
}
```

```
class ClassB {
    void accessMethod() {
        ClassA a=new ClassA();
        a.isPrivate=10;    // 非法
        a.privateMethod(); // 非法
    }
}
```

(3) protected 受保护变量可被声明它的类和派生的子类,以及同一个包中的类访问。这就像一个大家庭,家庭成员的秘密可被其他成员分享,也包括一些亲朋好友,但不想让外界知道。

受保护变量对子类的访问有一定的限制。当子类 and 父类在同一个包时,子类访问父类的受保护变量没有问题。如果子类处于其他包中,子类的对象可访问父类的受保护变量,但子类中由父类产生的对象就不能访问。

(4) package 包变量在声明时常常省略 package 关键字,即没有修饰符的成员被视为包成员。包成员可被声明它的类和同一个包中的其他类(包括派生子类)所访问,在其他包中的子类则不能访问父类的包成员。这就像值得信任的好朋友可以分享你的秘密,却不想让外地的家庭成员知道。

## 2. static 静态变量

static 声明的成员变量被视为类的成员变量,而不把它当作实例对象的成员变量。换句话说,静态变量是类固有的,可以直接引用,其他成员变量仅仅被声明,生成实例对象后才存在,才可以被引用。基于这样的事实,也把静态变量称为类变量,非静态变量称为实例变量。相应地,静态方法称为类方法,非静态方法称为实例方法。

例 4.4 静态变量的使用,运行结果如图 4.3 所示。

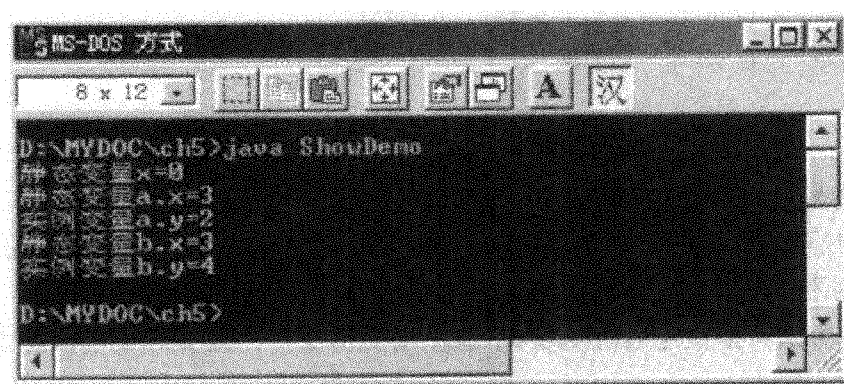


图 4.3

```
class StaticDemo {  
    static int x;  
    int y;  
  
    static public int getX() {  
        return x;  
    }  
    static public void setX(int newX) {  
        x = newX;  
    }  
    public int getY() {  
        return y;  
    }  
    public void setY(int newY) {  
        y = newY;  
    }  
}  
  
public class ShowDemo {
```

```

public static void main(String[] args) {
    System.out.println("静态变量 x=" + StaticDemo.getX());
    System.out.println("实例变量 y=" + StaticDemo.getY()); // 非法,编译时将出错
    StaticDemo a= new StaticDemo();
    StaticDemo b= new StaticDemo();

    a.setX(1);
    a.setY(2);
    b.setX(3);
    b.setY(4);

    System.out.println("静态变量 a. x=" + a.getX());
    System.out.println("实例变量 a. y=" + a.getY());
    System.out.println("静态变量 b. x=" + b.getX());
    System.out.println("实例变量 b. y=" + b.getY());
}
}

```

从输出结果中我们可以得出以下几点结论:

(1) 类的静态变量可以直接引用,例如在程序中我们使用了 `StaticDemo.getX()`,而非静态变量则不行。类的静态变量相当于某些程序语言的全局变量。

(2) 静态方法只能使用静态变量,不能使用实例变量。因为对象实例化之前,实例变量不可用。

(3) 类的静态变量只有一个版本,所有实例对象引用的都是同一个版本。例如程序中先用 `a.setX(1)` 为变量 `a.x` 赋值 1,后用 `b.setX(3)` 为变量 `b.x` 赋值 3,因它们使用的是同一个静态变量 `x`,所以 `a.getX()` 和 `b.getX()` 的返回值都是 3。

(4) 对象实例化后,每个实例变量都被制作了一个副本,它们之间互不影响。例如程序中用 `a.setY(2)` 为 `a.y` 赋值 2,后用 `b.setY(4)` 为 `b.y` 赋值 4,因它们使用的是实例变量 `y` 的不同副本,所以 `a.getY()` 和 `b.getY()` 的返回值分别是 2 和 4。

### 3. final 最终变量

一旦成员变量被声明为 `final`,在程序运行中将不能被改变。这样的成员变量就是一个常量。例如:

```
final double PI=3.14159;
```

该语句声明一个常量 `PI`,如果在后面试图重新对它赋值,将产生编译错误。另外,常量名一般用大写字母。常量和直接量一样不占用内存空间。

### 4. transient 过渡变量

Java 语言手册目前对 `transient` 修饰符没有明确说明,它一般用在对象序列化(object serialization)上,说明成员变量不许被序列化。

## 5. volatile 易失变量

volatile 声明的成员变量为易失变量,用来防止编译器对该成员进行某种优化。这是 Java 语言的高级特性,仅被少数程序员使用。

## 4.3 成员方法

对象的行为由方法实现,其他对象可以调用一个对象的方法,通过消息的传递实现对该对象行为的控制。下面我们讨论如何通过方法影响对象的行为。

### 4.3.1 成员方法的设计

类的设计集中体现在成员方法的设计上。良好的设计可以使类更加强壮,功能更加完善。成员方法的设计应该从类的整体行为出发,能正确响应外部消息,自然地改变对象的状态,并符合相对独立性、结构清晰、可重用性强等编程要求。我们来看下面的例子:

例 4.5 方法对对象行为的影响,运行结果见图 4.4。

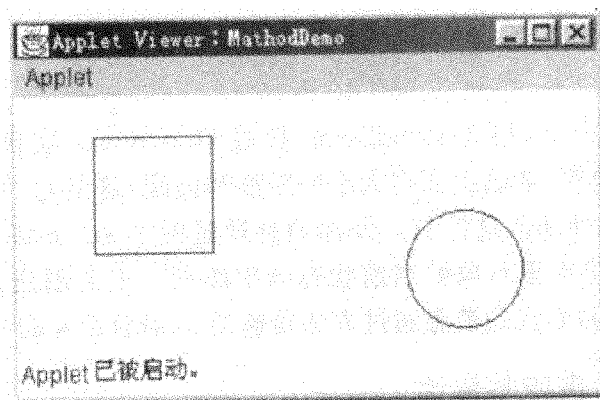


图 4.4

```
import java.awt.*;  
import java.applet.*;  
  
class DrawShape {  
    private int x, y, shape;  
    public void setPos(int xPos, int yPos) {  
        x=xPos;  
        y=yPos;  
    }  
    public void setShape(int choice) {  
        shape=choice;  
    }  
    public void draw(Graphics g) {
```

```

        if (shape == 1)
            g.drawRect(x, y, 60, 60);
        else if (shape == 2)
            g.drawOval(x, y, 60, 60);
        else
            g.drawString("形状参数不对!", 20, 120);
    }
}

```

```

public class MathodDemo extends Applet {
    final int BOX=1, OVAL=2;
    DrawShape a=new DrawShape();
    public void paint(Graphics g) {
        a.setPos(40,20);
        a.setShape(BOX);
        a.draw(g);
        a.setPos(200,60);
        a.setShape(OVAL);
        a.draw(g);
    }
}

```

例 4.5 设计了两个类,用 MathodDemo 控制 DrawShape 实例对象的行为。DrawShape 被设计成一个黑箱,能画出固定大小的矩形和椭圆,这是它的已知功能。画什么图形、画在什么地方则由外部消息控制。外部消息通过两个 set 方法改变了对象的位置和形状数据,draw 方法将根据对象的当前状态画出图形。主类创建了 DrawShape 的实例对象 a,并将图形位置和形状的消息通过方法传递给 a,最后由 a 的 draw 方法画出图形。

#### 4.3.2 成员方法的声明与修饰

成员方法相当于其他语言的函数或过程,是命令语句的集合。成员方法的声明一般放在成员变量的声明之后,声明语句的格式和顺序如下:

```

[public] [private] [protected] [package] // 访问控制修饰符
[static] [final] [abstract] [native] [synchronized]
返回值类型 方法名(参数表)[throws 异常类型]

```

4 种访问控制修饰符和成员变量的修饰符有相同作用,static 修饰符亦如此。下面主要介绍有不同含义的修饰符。

##### 1. final 最终方法

方法被声明为最终方法后,将不能被子类覆盖,即最终方法能被子类继承和使用但不能在子类中修改或重新定义它。这种修饰可以保护一些重要的方法不被修改,尤其是那些对类的状态和行为有关键性作用的方法被保护以后,可以避免未知情况的发生。



在 OOP 中,子类可以把父类的方法重新定义,使之具有新功能但又和父类的方法同名、同参数、同返回值,这种情况称为方法覆盖(override)。

有时不便于把整个类声明为最终类,这种保护太严格,不利于编程,此时可以有选择地把一些方法声明为最终方法,同样可以起到保护作用。

## 2. abstract 抽象方法

一个抽象类可以含有抽象方法。所谓抽象方法是指不能实现的方法,因为它没有方法体,所以抽象方法不能出现在非抽象类中。

为什么要使用抽象类和抽象方法呢?一个抽象类可以定义一个统一的编程接口,使其子类表现出共同的状态和行为,但各自的细节是不同的。子类共有的行为由抽象类中的抽象方法来约束,而子类行为的具体细节则通过抽象方法的覆盖来实现。这种机制可增加编程的灵活性,也是 OOP 继承树的衍生基础。

例如,直线、圆和矩形等图形对象都有一些共同的状态(位置,边界)和行为(移动、改变大小、画出)。你可以利用这些共性,把它们声明为同一个父类的子类。但是一个圆和一个矩形在很多细节方面又是不同的,父类不能把这些都包办代替,这时,一个抽象父类是最佳选择。首先声明一个抽象类 GraphicObject,提供所有子类可共享的成员变量和成员方法,然后声明一个抽象的画出方法 draw。子类继承父类后,通过覆盖 draw 实现画出自己的方法。可以这样做:

```
abstract class GraphicObject {
    int x, y, width, height;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    void setSize(int sizeW, int sizeH) {
        ...
    }
    abstract void draw();
}
```

GraphicObject 的每一个非抽象子类如 Circle 和 Rectangle,都要通过实现 draw 方法来画出自己:

```
class Circle extends GraphicObject {
    void draw() {
        ...
    }
}

class Rectangle extends GraphicObject {
    void draw() {
```

...  
}  
}  
一个抽象类不一定非要包含一个抽象方法,但一个类如果包含一个抽象方法就必须声明为抽象类。一个子类如果没有实现父类中的抽象方法也必须声明为抽象类。

### 3. native 本地方法

用其他语言编写的方法在 Java 中称为本地方法。如果你有一个用其他语言如 C++ 写成的函数库,希望在 Java 中能利用这些资源,你可以编写本地方法。

JDK 提供了 Java 本地接口 JNI(Java Native Interface),使得 Java 虚拟机能运行嵌入在 Java 程序中的其他语言的代码。这些语言包括 C/C++、FORTRAN、汇编语言等等。

嵌入外部语言代码出于以下几点考虑:Java 系统包不提供对平台依赖性程序的支持时;想利用现有的其他语言资源时;出于运行速度的要求而使用其他语言开发的运行模块。

下面这段程序实现了和 C 语言的接口:

```
class HelloWorld {  
    public native void displayHelloWorld();  
    static {  
        System.loadLibrary("hello");  
    }  
    public static void main(String[] args) {  
        new HelloWorld().displayHelloWorld();  
    }  
}
```

displayHelloWorld 方法被声明为本地方法,剩下的任务是用 C 语言编写程序实现这个方法,并用 C 语言编译成动态链接库文件 hello.DLL。

### 4. synchronized 同步方法

同步方法用于多线程编程。多线程在运行时,可能会同时存取一个数据。为避免数据的不一致性,应将方法声明为同步方法,对数据进行加锁,以保证线程的安全。详细内容请参考后面章节。

### 5. throws 异常类型

程序在运行时可能发生异常现象。每一个异常都对应着一个异常类,如果希望方法忽略某种异常,可将其抛出,使程序得以继续运行。我们在第 3 章介绍的例 3.8 就使用了抛出异常 throws IOException(输入/输出异常),以便顺利从键盘上读取字符。有关异常处理的详细内容请参考后面章节。

## 6. 返回值类型

Java 要求一个方法必须声明它的返回值类型。如果方法没有返回值就用关键字 `void` 作为返回值类型, 否则应使用基本数据类型或对象类型说明返回值类型, 如下面的语句:

```
public int getX();  
void setXY(int x, int y);  
public String getName();  
protected Object clone();
```

其中, `getX` 的返回值为 `int` 类型, `setXY` 没有返回值, `getName` 的返回值是 `String` 类。而 `clone` 的返回值则为 `Object` 类。

## 7. 方法名

方法名可以是任何有效的 Java 标识符。方法名可以和成员变量同名, 也可以和成员方法同名。同一个类中的方法同名现象在 OOP 中称为方法重载 (overload), 重载使一个类的多个方法享有同一个名称, 可以减轻程序员的负担。例如:

```
class DataPrint {  
    void print(String s) {  
        ...  
    }  
    void print(int i) {  
        ...  
    }  
    void print(float f) {  
        ...  
    }  
}
```

在非 OOP 语言中, 你需要定义 3 个不同名称的函数 `printString`、`printInt`、`printFloat` 才能实现 3 种类型的数据输出。在 Java 中, 你只要记住一个方法名就行了, 无论向 `print` 方法传递哪一种参数, 都能得到正确的输出。

同名方法由参数个数和类型来区分, 同一个类中不能有同名、同参数个数、同参数类型的方法出现。

## 8. 参数表

方法的调用者正是通过参数表将外部消息传递给方法加以处理的。在参数表中要声明参数的类型, 并用逗号分隔多个参数。

### 4.3.3 方法体

方法体包含在一对大括号中, 即使方法体中没有语句, 一对大括号也是必不可少的。

下面这段程序是一个处理格式化字符串的方法：

```
public String toString(int[] arr) {
    int i, n=arr.size();
    StringBuffer s=new StringBuffer();

    s.append("[");
    for (i=0; i<n; i++) {
        s.append(Integer.toString(arr[i]));
        if (i<n-1)
            s.append(",");
    }
    s.append("]");

    return s.toString();
}
```

在方法体中,可以声明多个变量,这些变量称为局部变量。程序中声明了3个局部变量,i和n为整型数,s为StringBuffer类的对象。局部变量的作用域为声明它的块,出了这个块,变量就消失了。

程序中的参数是一个整型数组arr,程序的目的是将数组元素转换成由方括号包起来的字符串。参数也属于局部变量,作用域是整个方法,出了方法,参数将消失。

可使用各种语句结构组成方法主体,实现方法的功能,这是最重要的。

程序中先将左方括号添加到s中,然后用一个循环将每一个数组元素转换成字符串添加到s。逗号的添加取决于数组元素的位置,最后一个数组元素后边就不再加逗号了。循环完毕,将右方括号添加到s,完成整个转换工作。

最后,如果有返回值,就用return语句返回给方法的调用者。

#### 4.3.4 消息传递

一个对象和外部交换信息主要靠方法的参数来传递。当然,如果允许的话,外部对象也可以直接存取一个对象的成员变量。出于整体考虑,大部分还是通过参数来传递。在Java中,可传递的参数包括任何数据类型,你已经见到了基本数据类型的参数传递、数组的传递和对象的传递。例4.2中就包含了对象g的传递,以下是程序片断:

```
class MyBox {
    private int x, y, width, height;
    ...
    public void draw(Graphics g) {
        g.drawRect(x, y, width, height);
    }
}
```

g是一个Graphics对象,它被传递到MyBox的draw方法中,然后调用g的

drawRect 方法画出图形。不像其他语言,Java 不能传递方法,但你可以传递一个对象,然后调用该对象的方法。

在其他语言中,函数调用或过程调用有传值调用和传地址调用之分。在 Java 中,参数传递是传值调用。调用方法时,如果传递的参数是基本数据类型,在方法中将不能改变参数的值,这意味着你只能使用它们。如果传递的是对象引用,你也不能在方法中修改这个引用,但可以调用对象的方法以及修改允许存取的成员变量。

所以,如果不想改变参数的值,可以采用传值调用的方法。如果想改变参数的值,可采用传递对象的方法,间接修改参数的值。请看下面的例子。

**例 4.6** 参数传递示例 1,运行结果如图 4.5 所示。

```
class PassDemol {
    public static void main(String[] args) {
        int x=10, y=10;
        doPower(x, y);
        System.out.println("x="+x+", y="+y);
    }

    static void doPower(int passX, int passY) {
        passX=passX * passX;
        passY=passY * passY;
        System.out.println("passX="+passX+", passY="+passY);
    }
}
```

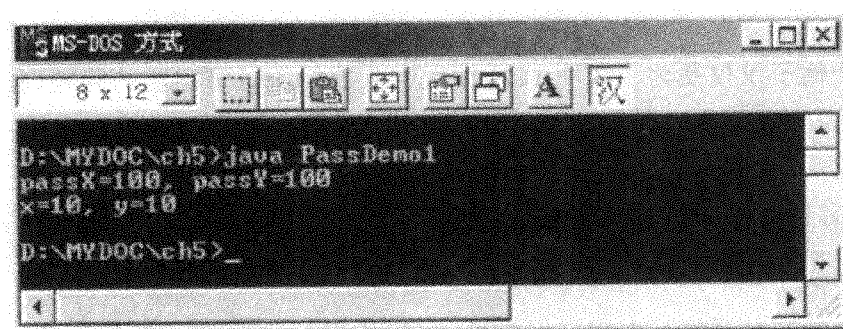


图 4.5

这段程序的原意是给 x 和 y 赋一个初值,然后调用方法 doPower 对 x 和 y 做乘方,再由系统输出 x 和 y 的乘方值。但这个程序得不到预期的结果,原因是 doPower 采用了传值调用。调用 doPower 时,将产生两个参数 passX 和 passY, x 和 y 的值被传递给这两个参数。尽管在方法中计算了参数的平方,但从 doPower 方法返回后,参数消失,此时 x 和 y 的值仍是初值。

改写后的例 4.7 可实现程序员的原意。

**例 4.7** 参数传递示例 2,运行结果如图 4.6 所示。

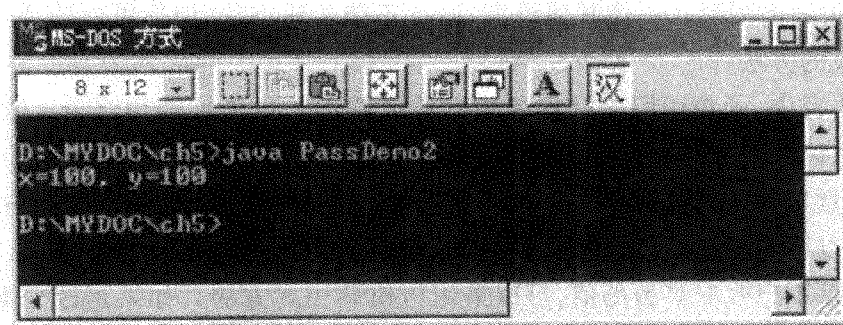


图 4.6

```
class PassDemo2 {
    public static void main(String[] args) {
        Power p=new Power();
        p.doPower(10,10);
        System.out.println("x="+p.x+", y="+p.y);
    }
}
```

```
class Power {
    int x, y;
    void doPower(int passX, int passY) {
        x=passX * passX;
        y=passY * passY;
    }
}
```

以上两个例子仅仅是为了说明 Java 编程中参数传递时要注意的问题,作为实用编程是不可取的,因为完全可以采用其他更好的方法来实现。

## 4.4 对象实例化

类是对象的产品图纸。在现实世界里,我们使用的是产品而不是产品的图纸。同样道理,Java 运行的是用类创建的实例化对象。一个典型的 Java 程序会创建很多对象,它们通过消息传递进行相互交流,共同完成程序的功能。一旦任务完成,对象就会被垃圾收集器收回,完成它从创建、使用到清除的生命三部曲。

### 4.4.1 创建对象

你必须拥有一个合适的类才能创建一个合适的对象,有了合适的对象才能完成合适的工作。下面的三条语句分别创建了三个对象:

```
Label label1=new Label("标签");
TextField field1=new TextField(10);
```

```
MyBox box1=new MyBox(20,20,100,100);
```

其中,Label 和 TextField 是系统类,MyBox 是一个自定义类。第一条语句创建了一个标签对象 label1,第二条语句创建了一个文本对象 field1,第三条语句创建了一个 MyBox 对象 box1。new 操作符用于对象的实例化,new 调用了类的构造方法初始化创建的对象,至此,每个对象就可以被使用了。

我们归纳出创建对象语句的三个组成部分:声明对象、实例化、初始化。

声明对象由类名和对象名组成,如等号左边部分;实例化由 new 操作符实现,实例化就是为对象分配内存。new 操作符需要一个参数,就是类的构造方法;初始化工作由构造方法完成。构造方法一方面提供了类名,由 new 操作符根据类名决定为新建对象分配多大的内存。另一方面,构造方法将新建对象初始化。

new 操作符返回一个引用或参考(reference),并将它赋给对象名。对象引用实际上是一个指针,指向对象所在的内存地址。但这个指针(即对象名)是无法更改的,因为出于安全性和健壮性的考虑,Java 取消了指针操作。

创建对象相当于定义一个变量,既可采用上述方法,也可以把上述步骤分开进行,即先声明对象,以后再创建。如:

```
Label label1;  
...  
label1=new Label("a label");
```

#### 4.4.2 使用对象

对象被实例化后即可使用。你可能希望用它做一些事情,对象提供了两种应用方式供选择。你可以直接存取对象的变量,也可以调用它的方法。

创建对象时,一个对象的所有变量和方法代码都被读到专为它开辟的内存区域中。为了让解释器知道代码的内存地址,使用对象的任何成员时都要加上引用。即在变量和方法的前面加上对象名,并用圆点分隔。例如:

```
String s= Label1.getText();  
label1.setText("new string");
```

还可以在使用时直接加入对象引用,例如:

```
Rectangle r=new Rectangle(10,10,100,100);  
...  
r.setLocation(new Point(50,50));
```

其中,new Point(50,50) 作为 setLocation 方法的参数被直接创建,并将 new 返回的引用传递给 setLocation 方法。这种情况称为类的匿名,因为 new 返回的引用没有和有关标识符联系起来,此时 new Point(50,50) 称为匿名类。

注意:从方法返回后,Point 对象将被垃圾收集器自动收回,因为这个新创建的对象引用没有保存到一个变量里。

### 4.4.3 清除对象

很多 OOP 语言要求程序员跟踪所创建的对象,当不再使用这些对象时,由程序员负责清除它们,收回所占用的内存。这是一件很头痛并经常出错的事情。

Java 引入了新的内存管理机制,由 Java 虚拟机担当垃圾收集器的工作。你可以任意创建对象而不用担心如何清除它们,垃圾收集器会自动清除它们。

使用 new 操作符创建对象后,Java 虚拟机自动为该对象分配内存并保持跟踪。Java 虚拟机能判断出对象是否还被引用,对不再被引用的对象释放其占用的内存。这种定期寻找不再使用的对象并自动释放对象占用内存的过程称为垃圾收集。Java 虚拟机实际上是利用变量生存期来管理内存的,对象的引用被保存在变量中,当程序跳出变量所在的区域后,它就会被自动清除。

如果要明确地清除一个对象,你可以自行清除它,只需把一个空值赋给这个对象引用即可。如:

```
Rectangle r=new Rectangle(10,10,100,100);  
...  
r=null;
```

上述语句执行后,r 对象将被清除。

Java 的垃圾收集机制大大减轻了程序员的负担,你不用再编写专门的内存回收程序解决内存分配问题。不仅提高了编程效率,而且进一步增强了 Java 程序的稳固性。

## 习 题

- 4-1 静态变量有何特点? 如何存取静态变量?
- 4-2 静态方法有何特点? 静态方法存取成员变量时有何要求?
- 4-3 何为抽象类、抽象方法?
- 4-4 类与对象有何关系?
- 4-5 Java 程序由什么构成? 程序设计的基本思想是什么?
- 4-6 什么是系统类、自定义类、父类、子类?
- 4-7 什么是类变量、成员变量、实例变量、局部变量?
- 4-8 分析下面这段程序中 MyBox 的各个成员。

```
class MyBox {  
    static int x, y;  
    int width, height;  
    MyBox() {x=0; y=0; width=0; height=0;}  
    public void setPosition (int xPos, int yPos) {x=xPos; y=yPos;}  
    public void setSize (int w, int h) {width=w; height=h;}  
    static int getX() {return x;}  
    static int getY() {return y;}  
}
```



```
protected void draw(Graphics g) {g.drawRect(x, y, width, height);}
}
```

- 4-9** 创建一个 Rectangle 类,添加两个属性 width、height。
- 4-10** 在 Rectangle 中添加两个方法计算矩形的周长和面积。
- 4-11** 编程利用 Rectangle 输出一个矩形的周长和面积。
- 4-12** 创建一个 Circle 类,具有 4 个属性 x、y、width、height。添加构造方法,画出方法,利用 fillOval 画椭圆。
- 4-13** 设计一个 Applet,利用 Circle 画出 20 个同心圆。
- 4-14** 设计一个 Array 类,添加一个整型数组,添加构造方法对数组赋初值。
- 4-15** 为 Array 类添加数组求和方法,添加返回求和值的方法。
- 4-16** 编程利用 Array 计算数组的值并输出。



# 第5章

## 类的继承性和多态性

OOP 的关键技术是继承和多态。继承实现了软件重用,多态使程序增加新功能变得容易。本章主要讨论类的继承性和多态性,以及一些相关问题。

### 5.1 类的继承

新类从现有的类中产生,保留了现有类的属性和方法并根据需要加以修改。新类还可添加新的属性和方法,这些新增功能允许以统一的风格处理不同类型的数据。这种现象就称为类的继承。

#### 5.1.1 父类和子类

当你建立一个新类时,不必写出全部成员变量和成员方法。只要简单地声明这个类是从一个已定义的类继承下来的,就可以使用被继承类的全部成员。被继承的类称为父类或超类(superclass),这个新类就是子类。

Java 提供了一个庞大的类库供你继承和使用。设计这些类是出于公用的目的,因此,很少有某个类恰恰满足你的需要。你必须设计自己的能处理实际问题的类,如果你设计的这个类仅仅实现了继承,则和父类毫无两样。所以,通常要对子类进行扩展,即添加新的属性和方法。这使得子类要比父类大,但更具特殊性,代表着一组更具体的对象。继承的意义就在于此。

在类的声明语句中加入 `extends` 关键字和指定的类名即可实现类的继承,例如:

```
public class MyApplet extends java.applet.Applet
public class MyApplication extends Frame
public class MyApp1 extends Object
public class MyApp2
```

第一条语句声明子类 `MyApplet` 的父类是 `Applet`,并指明 `Applet` 的层次结构;第二条语句声明子类 `MyApplication` 的父类是 `Frame`;第三条语句声明子类 `MyApp1` 的父类是 `Object`,但编程中通常会省略 `extends Object` 子句;第四条语句在字面上没有 `extends`,

但实际上等价于 `public class MyApp2 extends Object`。

那么,类的继承是从什么地方开始的? 又是如何延续下来的呢? 我们看图 5.1 所示的模拟图。

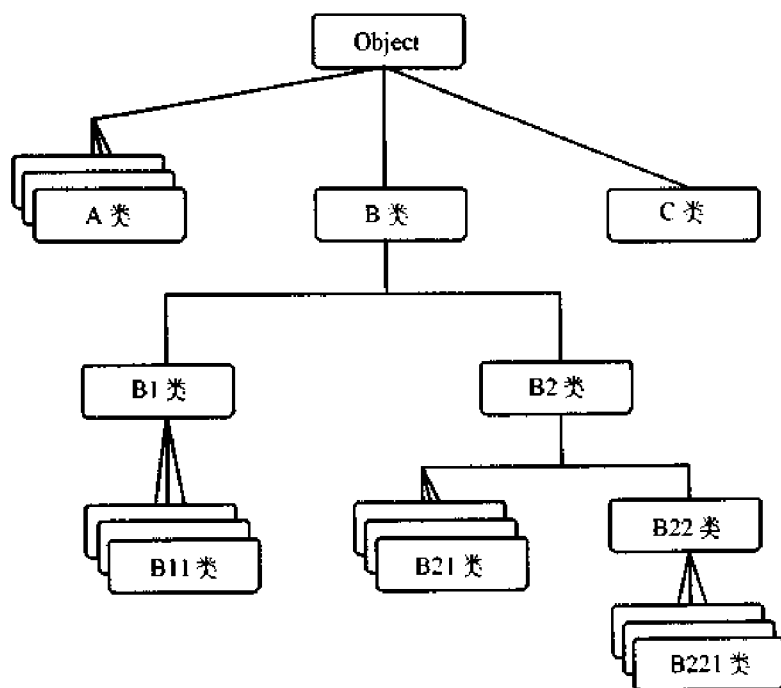


图 5.1

图 5.1 反映了 Java 类的层次结构。最顶端的类是 `Object`, 它在 `java.lang` 中定义, 是所有类的始祖。一个类可以有多个子类, 也可以没有子类, 但它必定有一个父类 (`Object` 除外)。

子类不能继承父类中的 `private` 成员, 除此之外, 其他所有的成员都可以通过继承变为子类的成员。另一方面, 对继承的理解应该扩展到整个父类的分支。也就是说, 子类继承的成员实际上是整个父系的所有成员。例如, `toString` 这个方法是在 `Object` 中声明的, 被层层继承了下来, 用于输出当前对象的基本信息。

至此, 我们得出如下结论: 子类只能有一个父类。尽管省略了 `extends`, 子类还是有一个父类 `Object`。子类继承了父类和祖先的成员, 可以使用这些成员。在需要的时候, 子类可以添加新的成员变量和方法, 也可以隐藏父类的成员变量或覆盖父类的成员方法。

### 5.1.2 成员变量的继承和隐藏

**例 5.1** 从点 `Point` 扩展到线 `Line` 和圆 `Circle` 的例子。

```
public class Point {
    protected int x, y;
    Point(int a, int b) {setPoint(a, b);}
    public void setPoint(int a, int b) {
        x=a;
```

# 第5章

## 类的继承性和多态性

OOP 的关键技术是继承和多态。继承实现了软件重用,多态使程序增加新功能变得容易。本章主要讨论类的继承性和多态性,以及一些相关问题。

### 5.1 类的继承

新类从现有的类中产生,保留了现有类的属性和方法并根据需要加以修改。新类还可添加新的属性和方法,这些新增功能允许以统一的风格处理不同类型的数据。这种现象就称为类的继承。

#### 5.1.1 父类和子类

当你建立一个新类时,不必写出全部成员变量和成员方法。只要简单地声明这个类是从一个已定义的类继承下来的,就可以使用被继承类的全部成员。被继承的类称为父类或超类(superclass),这个新类就是子类。

Java 提供了一个庞大的类库供你继承和使用。设计这些类是出于公用的目的,因此,很少有某个类恰恰满足你的需要。你必须设计自己的能处理实际问题的类,如果你设计的这个类仅仅实现了继承,则和父类毫无两样。所以,通常要对子类进行扩展,即添加新的属性和方法。这使得子类要比父类大,但更具特殊性,代表着一组更具体的对象。继承的意义就在于此。

在类的声明语句中加入 `extends` 关键字和指定的类名即可实现类的继承,例如:

```
public class MyApplet extends java.applet.Applet
public class MyApplication extends Frame
public class MyApp1 extends Object
public class MyApp2
```

第一条语句声明子类 `MyApplet` 的父类是 `Applet`,并指明 `Applet` 的层次结构;第二条语句声明子类 `MyApplication` 的父类是 `Frame`;第三条语句声明子类 `MyApp1` 的父类是 `Object`,但编程中通常会省略 `extends Object` 子句;第四条语句在字面上没有 `extends`,

```

Line           // 线的构造方法
setLine        // 设定线的两个端点坐标值
getX, getY     // 返回起点坐标 x 和 y 的值
getEndX, getEndY // 返回终点坐标 endX 和 endY 的值
length         // 返回线的长度
x, y           // 继承父类的受保护成员变量,但被子类隐藏
setPoint       // 继承父类的方法
getX, getY     // 继承父类的方法,但被子类覆盖
Circle:
radius         // 子类受保护的成员变量,代表圆的半径
Circle         // 圆的构造方法
setRadius      // 设定半径值
getRadius      // 返回半径值
area           // 返回圆面积
x, y           // 继承父类的受保护成员变量
setPoint       // 继承父类的方法
getX, getY     // 继承父类的方法

```

从上面的分析可以看到,子类 Circle 继承了父类 Point 中的成员变量 x 和 y,也继承了 setPoint、getX 和 getY 方法。添加了自己的成员变量 radius,成员方法 setRadius、getRadius 和 area。在构造方法 Circle 中,调用了父类的构造方法设定圆心坐标(父类的构造方法必须处于第一行)。

子类 Line 隐藏了父类 Point 中的两个成员变量 x 和 y,所谓隐藏是指子类重新定义了父类中的同名变量。子类执行自己的方法时,操作的是子类的变量,子类执行父类的方法时,操作的是父类的变量。在子类中要特别注意成员变量的命名,防止无意中隐藏了父类的关键成员变量,这有可能给你的程序带来麻烦。

Line 还覆盖了 Point 的两个方法 getX 和 getY。

### 5.1.3 成员方法的覆盖

方法的覆盖(override),为子类提供了修改父类成员方法的能力。例如,子类可以修改层层继承下来的 toString 方法,让它输出一些更有用的信息。下面的例子显示了在子类 Circle 中添加 toString 方法,用来返回圆半径和圆面积信息。

**例 5.2** 对 Object 的 toString 方法的覆盖。结果如图 5.2 所示。

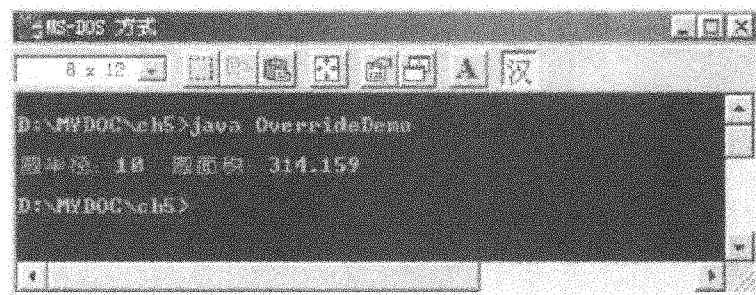


图 5.2 对 Object 的 toString 方法的覆盖

```

class Circle {
    private int radius;
    Circle(int r) {setRadius(r);}
    public void setRadius(int r) {radius=r;}
    public int getRadius() {return radius;}
    public double area() {return 3.14159 * radius * radius;}
    public String toString() {
        return "圆半径:" + getRadius() + " 圆面积:" + area();
    }
}

public class OverrideDemo {
    public static void main(String args[]) {
        Circle c=new Circle(10);
        System.out.println("\n" + c.toString());
    }
}

```

程序中改写了上一节介绍的 Circle, 添加了 toString 方法并修改了它的返回值。由于 toString 和继承下来的方法同名、同返回值类型, 因此就覆盖了父类中的 toString。而父类 Point 中的 toString 方法尽管没有显示出来, 它仍然存在, 是从 Object 中继承下来的。

方法覆盖时要特别注意: 用来覆盖的子类方法应和被覆盖的父类方法保持同名、同返回值类型, 以及相同的参数个数和参数类型。如果被覆盖的方法没有声明抛出异常, 子类的覆盖方法可以有不同的抛出异常子句。

有时, 可能不需要完全覆盖一个方法, 你可以部分覆盖一个方法。部分覆盖是在原方法的基础上添加新的功能, 即在子类的覆盖方法中添加一条语句: super. 原方法名, 然后加入其他语句。

注意: 不能覆盖父类中的 final 方法, 因为设计这类方法的目的是为了防止覆盖。同样也不能覆盖父类中的 static 方法, 但可以隐藏这类方法。也就是说, 在子类中声明的同名静态方法实际上隐藏了父类中的静态方法。此外, 子类必须覆盖父类中的抽象方法。

#### 5.1.4 this 和 super

this 和 super 有什么作用? 在什么情况下使用它们? 我们看下面的例子。

例 5.3 this 和 super 的使用。运行结果如图 5.3 所示。

```

class Point {
    protected int x, y;
    Point(int a, int b) {setPoint(a, b);}
    public void setPoint(int a, int b) {
        x=a;
        y=b;
    }
}

```

```

    }

    class Line extends Point {
        protected int x, y;
        Line(int a, int b) {
            super(a, b);
            setLine(a, b);
        }

        public void setLine(int x, int y) {
            this.x = x + x;
            this.y = y + y;
        }

        public double length() {
            int x1 = super.x, y1 = super.y, x2 = this.x, y2 = this.y;
            return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
        }

        public String toString() {
            return "直线端点:[" + super.x + "," + super.y + "]" [" +
                x + "," + y + "] 直线长度:" + this.length();
        }
    }

    public class ThisDemo {
        public static void main(String args[]) {
            Line line = new Line(50, 50);
            System.out.println("\n" + line.toString());
        }
    }

```

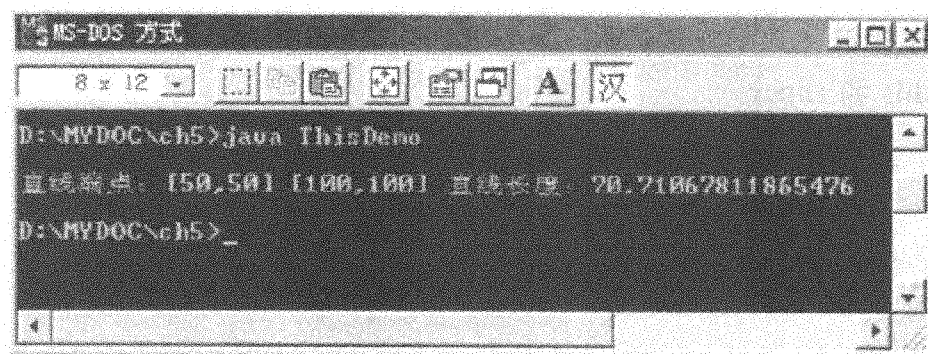


图 5.3

例 5.3 介绍了 this、super 和 super() 的使用。Point 中定义了成员变量 x 和 y, Line



中也定义了成员变量 `x` 和 `y`, 而构造方法 `Line()` 中也用 `x` 和 `y` 作为参数, 这种变量同名多次引发了变量的隐藏。首先是 `Line` 中的 `x` 和 `y` 隐藏了 `Point` 中的 `x` 和 `y`, 然后 `setLine` 方法中的参数 `x` 和 `y` 又隐藏了 `Line` 的 `x` 和 `y`。使用这些变量时, 如果分不清它们是属于谁的, 就会带来混乱。

`Point` 中已经有了两个变量表示点的位置, `Line` 只需再定义两个变量表示另一个点的位置就足够了。使用时, 可以用这 4 个变量来表示一条直线的两个端点。因此, 我们在构造方法中将代表初值的两个参数分别使用; 通过 `super(a,b)` 调用父类的构造方法为父类的 `x` 和 `y` 赋值; 通过 `setLine(a,b)` 为子类的 `x` 和 `y` 赋值。

在 `setLine` 方法中, 因为参数名和成员变量名相同, 如果为成员变量赋值, 就必须加上 `this` 引用, 告诉编译器是为当前类的成员变量赋值。同理, 在 `length` 和 `toString` 方法中使用父类成员变量时, 就必须加上 `super` 引用, 告诉编译器使用的是父类的成员变量。

设计这个程序的目的是为了说明 `this` 和 `super` 的用法, 如果不用变量隐藏, 或者在子类中定义足够多的成员变量, 则不需要 `this` 和 `super`, 但这样就发挥不了继承的优势。

总结上例, 我们概括出 `this` 和 `super` 的作用:

(1) `this` 实际代表的是当前类或对象本身。在一个类中, `this` 表示对当前类的引用, 在使用类的成员时隐含着 `this` 引用, 尽管可以不明确地写出, 例如 `length` 和 `toString` 中对 `x` 和 `y` 的使用。当一个类被实例化为一个对象时, `this` 就是对象名的另一种表示。通过 `this` 可顺利地访问对象, 凡在需要使用对象名的地方均可用 `this` 代替。例如:

```
public class Demo extends java.applet.Applet implements ActionListener {
    Button button1=new Button("Add");
    ...
    public void init() {
        ...
        add(button1);
        button1.addActionListener(this); // 等价于 button1.addActionListener(Demo);
    }
    ...
}
```

(2) `super` 代表着父类。如果子类的变量隐藏了父类的变量, 使用不加引用的变量一定是子类的变量, 如果使用父类的变量, 就必须加上 `super` 引用。同样道理, 如果有方法覆盖的发生, 调用父类的方法时也必须加上 `super` 引用。考虑下面两个类:

```
class ClassA {
    boolean var;
    void method() {
        var=true;
    }
}

class ClassB extends ClassA {
```

```

boolean var;
void method() {
    var = false;
    super.method();
    System.out.println(var);
    System.out.println(super.var);
}
}

```

ClassB 隐藏了 ClassA 的成员变量和方法,将 var 赋值为 false,调用 super.method() 则将 ClassA 的 var 赋值为 true,因此输出结果分别是: false 和 true。

(3) super() 可用来调用父类的构造方法。这也是唯一由程序员间接调用类的构造方法的途径,因为 Java 规定类的构造方法只能由 new 操作符调用,程序员不能直接调用。同理,this() 可用来间接调用当前类或对象的构造方法。

## 5.2 类的多态

类的继承发生在多个类之间,而类的多态只发生在一个类上。在一个类中,可以定义多个同名的方法,只要保持它们的参数个数和类型不同。这种现象就称为类的多态。

多态使程序简洁,为程序员带来很大便利。在 OOP 中,当程序要实现多个相近的功能时,就给相应的方法起一个共同的名字,用不同的参数代表不同的功能。这样,在使用方法时不论传递什么参数,只要能被程序识别就可以得到确定的结果。

类的多态性体现在方法的重载(overload)上,包括成员方法和构造方法的重载。

### 5.2.1 成员方法的重载

方法的重载是指对同名方法的重新定义。我们看下面这个例子。

**例 5.4** 对不同的数进行排序输出,运行结果见图 5.4。

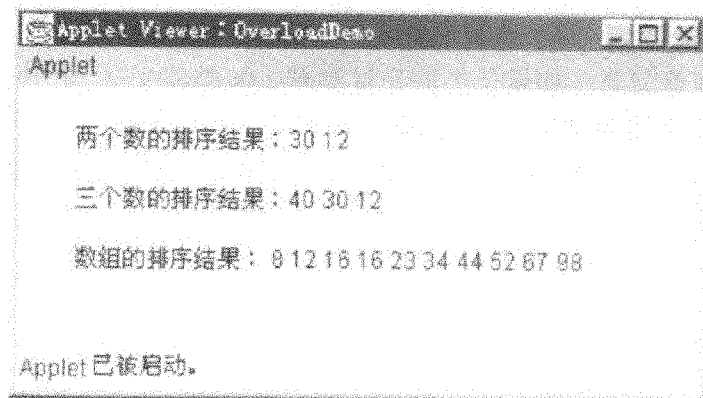


图 5.4

```
import java.awt.Graphics;
```

```
import java.applet.Applet;
```

```
class IntSort {
```

```
    public String sort(int a, int b) {
```

```
        if (a>b)
```

```
            return a+" "+b;
```

```
        else
```

```
            return b+" "+a;
```

```
    }
```

```
    public String sort(int a, int b, int c) {
```

```
        int swap;
```

```
        if (a<b) {
```

```
            swap=a;
```

```
            a=b;
```

```
            b=swap;
```

```
        }
```

```
        if (a<c) {
```

```
            swap=a;
```

```
            a=c;
```

```
            c=swap;
```

```
        }
```

```
        if (b<c) {
```

```
            swap=b;
```

```
            b=c;
```

```
            c=swap;
```

```
        }
```

```
        return a+" "+b+" "+c;
```

```
    }
```

```
    public String sort(int arr[]) {
```

```
        String s="";
```

```
        int swap;
```

```
        for (int i=0; i<arr.length; i++)
```

```
            for (int j=0; j<arr.length-1; j++)
```

```
                if (arr[j]>arr[j+1]) {
```

```
                    swap=arr[j];
```

```
                    arr[j]=arr[j+1];
```

```
                    arr[j+1]=swap;
```

```
                }
```

```
        for (int i=0; i<arr.length; i++)
```

```
            s=s+arr[i]+" ";
```

```
        return s;
```

```

    }
}

public class OverloadDemo extends Applet {
    IntSort s=new IntSort();
    public void paint(Graphics g) {
        int a=30, b=12, c=40;
        int arr[]={34,8,12,67,44,98,52,23,16,16};
        g.drawString("两个数的排序结果:" +s.sort(a,b),30,30);
        g.drawString("三个数的排序结果:" +s.sort(a,b,c),30,60);
        g.drawString("数组的排序结果:" +s.sort(arr),30,90);
    }
}

```

程序可实现对两个数、三个数和数组的排序。在主类 OverloadDemo 中,分别向同一个方法 sort 传递三种参数,都得到了正确的输出。

IntSort 中没有添加成员变量,仅仅提供了三个同名方法:

```

public String sort (int a, int b)
public String sort (int a, int b, int c)
public String sort (int arr[])

```

它们的区别在于参数的个数和类型的差别,Java 解释器在运行这个程序时,可以根据参数的不同来调用不同的方法。

这种多态性使类能够向外提供一个较为一致的接口,对程序员来说,不必关心同名方法内部的细节差别,只要掌握它们在使用时要求什么参数就可以了。

你应该注意到,程序中调用了 IntSort 的构造方法,但这个构造方法并没有在 IntSort 中出现,它是怎么来的呢?

## 5.2.2 构造方法的重载

重载构造方法的目的是提供多种初始化对象的能力,使程序员可以根据实际需要选用合适的构造方法来初始化对象。下面我们看一个例子。

**例 5.5** 构造方法的重载。运行结果如图 5.5 所示。

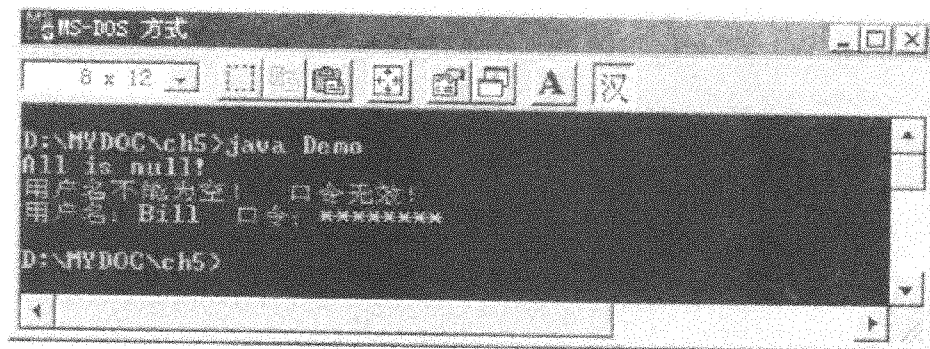


图 5.5

```

class RunDemo {
    private String userName, password;

    RunDemo() {
        System.out.println("All is null!");
    }

    RunDemo(String name) {
        userName=name;
    }

    RunDemo(String name, String pwd) {
        this(name);
        password=pwd;
        check();
    }

    void check() {
        String s=null;
        if (userName!= null)
            s="用户名："+userName;
        else
            s="用户名不能为空!";
        if (password!= "ThisWord")
            s=s+" 口令无效!";
        else
            s=s+" 口令：* * * * *";
        System.out.println(s);
    }
}

public class Demo {
    public static void main(String[] args) {
        new RunDemo();
        new RunDemo("Bill");
        new RunDemo(null, "Bill");
        new RunDemo("Bill", "ThisWord");
    }
}

```

例 5.5 有三个构造方法,其中无参构造方法 RunDemo() 的实际作用是对成员变量赋默认初值,由于 userName 和 password 都是 String 类,所以它们的默认初值为 null。第二个构造方法 RunDemo(String) 只有一个参数,用来对成员变量 userName 赋初值。在第三个构造方法 RunDemo(String, String) 中你看到了更多的内容,首先调用 this

(name), 其实际作用就是调用当前类的构造方法 `RunDemo(String name)`; 然后对成员变量 `password` 赋值; 最后调用 `check` 方法来检查 `userName` 和 `password`, 类似于一般程序的口令验证。

重载构造方法的执行由类根据实际参数的个数、类型和顺序确定, 为程序员提供了较大的灵活性。在例 5.4 中, 注意到程序调用了 `IntSort` 的构造方法, 但这个构造方法并没有在 `IntSort` 中出现, 它是怎么来的呢? 现在回答这个问题。

每一个类都有一个默认的构造方法, 这就是和类同名的无参构造方法。它实际上是父类的构造方法, 创建子类时由父类自动提供。类的构造方法是不能继承的, 因为构造方法不是类的成员, 没有返回值, 也不需要修饰符。又由于父类的构造方法和父类同名, 在子类中继承父类的构造方法肯定和子类不同名, 这样的继承是无意义的。但子类可以调用父类的构造方法, 如在例 5.3 中:

```
class Line extends Point {
    protected int x, y;
    Line(int a, int b) {
        super(a, b);
        setLine(a, b);
    }
    ...
}
```

其中 `super(a, b)` 就是在子类中调用父类的构造方法, 为父类的成员变量赋初值。

注意: `super()` 只能出现在子类的构造方法中, 而且必须是子类构造方法中的第一条可执行语句。

## 5.3 进一步讨论的问题

在前面两节我们讨论了 Java 的继承性和多态性, 还有一些有关这两种特性的问题需要进一步讨论, 这就是对象的克隆、子类对象和父类对象的关系以及类的包容。

### 5.3.1 对象的克隆

在 Java 中可以实现对象的克隆, 即从现存的对象复制出一个完全一样的副本。克隆由方法 `clone` 实现, 它是从 `Object` 继承下来的。下面是一个例子。

**例 5.6** 对象的克隆, 运行结果如图 5.6 所示。

```
import java.awt.*;
import java.applet.Applet;

class DrawOval implements Cloneable {
    int x, y, width, height;
```

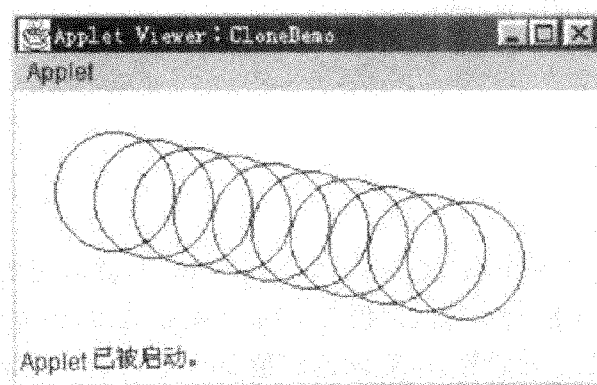


图 5.6

```

public void setPos(int x1, int y1) {
    x=x1;
    y=y1;
}

public void setOval(int w, int h) {
    width=w;
    height=h;
}

public void draw(Graphics g) {
    g.drawOval(x, y, width, height);
}

protected Object clone() {
    try {
        DrawOval clonedObject=(DrawOval)super.clone();
        return clonedObject;
    }
    catch (CloneNotSupportedException e) {
        throw new InternalError();
    }
}

public class CloneDemo extends Applet {
    public void paint(Graphics g) {
        DrawOval c[]=new DrawOval[10];
        DrawOval a=new DrawOval();
        a.setPos(20,20);
        a.setOval(60,60);
        for (int i=0; i<10; i++) {

```

```

        c[i] = (DrawOval)a.clone();
        c[i].setPos(20+i*20,20+i*4);
        c[i].draw(g);
    }
}
}

```

DrawOval 是 Object 的子类, 实现了 Cloneable 接口, 并对父类的 clone 方法进行了覆盖。clone 声明为受保护的并且返回值为 Object 类型的方法, 方法体中用 try...catch 来捕捉克隆过程中可能出现的异常, 它的返回值是一个对象。DrawOval 能在给定位置以给定大小画出一个椭圆, 由它产生的对象可以被克隆。主类中首先创建了 DrawOval 的对象 a, 声明一个数组用来保存从 a 克隆的对象, 然后将这些复制品在不同位置上画出来。克隆方法使用了类型转换, 请读者思考它的作用。

注意: 克隆一个对象时, 首先要检查它是否能被克隆。如果这个对象实现了 Cloneable 接口, 它就可以被克隆, 否则将抛出一个异常 CloneNotSupportedException。Object 类本身没有实现这个接口, 因此, 它的直接子类必须实现 Cloneable 接口才能使对象被克隆。克隆出来的对象和原对象是同一种类型, 它的成员变量被初始化为相同的值。

### 5.3.2 子类对象和父类对象的关系

我们知道, 子类和父类存在继承关系, 那么子类对象和父类对象存在什么关系呢? 它们有如下关系: 子类对象可被看作相应父类的对象。也就是说, 尽管从父类派生出来的子类千差万别, 但由这些子类产生的对象可以自动成为父类的对象。反之则不然, 父类对象不能自动成为子类的对象, 但在一定条件下, 它们可以互相转换。

**例 5.7** 父类对象和子类对象的转换。运行结果如图 5.7 所示。

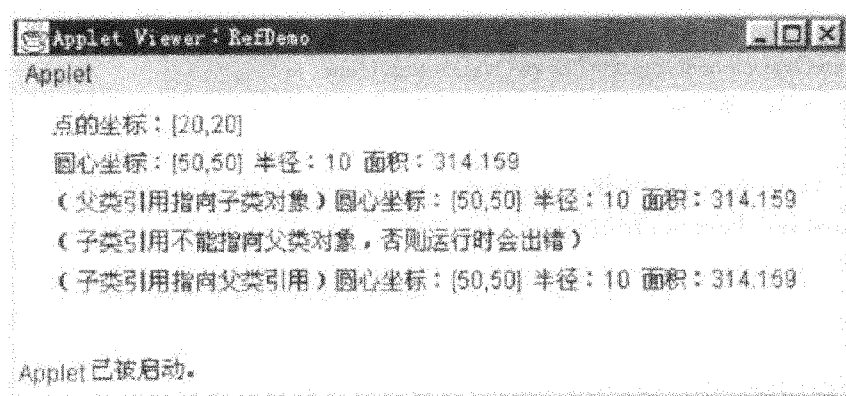


图 5.7 父类对象和子类对象的转换

```

import java.awt. * ;
import java.applet. Applet;

```

```

class Point {
    int x, y;

```



```

Point(int a, int b) {
    setPoint(a,b);
}

public void setPoint(int a, int b) {
    x=a;
    y=b;
}

public String toString() {
    return "点的坐标:["+x+","+y+"]";
}
}

class Circle extends Point {
    int radius;
    Circle(int a, int b, int r) {
        super(a,b); setRadius(r);
    }

    public void setRadius(int r) {
        radius=r;
    }

    public double area() {
        return 3.14159 * radius * radius;
    }

    public String toString() {
        return "圆心坐标:["+x+","+y+"] 半径:"+radius+" 面积:"+area();
    }
}

public class RefDemo extends Applet {
    Point pointRef, p=new Point(20,20);
    Circle circleRef, c=new Circle(50,50,10);

    public void paint(Graphics g) {
        g.drawString(p.toString(),20,20);
        g.drawString(c.toString(),20,40);
        pointRef=c;
        g.drawString("(父类引用指向子类对象)" + pointRef.toString(),20,60);
        // circleRef=p; // 错误引用,编译时不能通过
    }
}

```

```

        g.drawString("(子类引用不能指向父类对象,否则运行时会出错)",20,80);
        circleRef=(Circle)pointRef;
        g.drawString("(子类引用指向父类引用)" + circleRef.toString(),20,100);
    }
}

```

在这个程序中,Circle 是 Point 的子类。我们在主类 RefDemo 中声明了两个父类引用 pointRef 和 p,将 p 指向新创建的父类对象。又声明了两个子类引用 circleRef 和 c,将 c 指向新创建的子类对象。

从程序中我们可以看到,父类引用 pointRef 可以转换成子类对象 c,而子类引用 circleRef 却不能转换成父类对象 p。只有当父类引用指向一个子类对象时,该父类引用才能转换成子类引用,如语句 circleRef=(Circle)pointRef 所示。

子类对象和父类对象的转换可以实现一些有用的操作。例如把多个子类产生的对象统一转换到一个父类对象数组里,每个对象数组元素指向不同的子类对象,这可为对象管理带来便利。

### 5.3.3 类的包容

在 Java 中,你可以把一个类声明为另一个类的成员,这样的类称为被包容的类(nested class),例如:

```

class EnclosingClass {
    ...
    static class NestedClass {
        ...
    }

    class InnerClass {
        ...
    }
}

```

类的包容可以加强类之间的联系,被包容的类可以无条件地使用包容它的类的所有成员,包括私有成员。被包容的类的作用域仅限于包容它的类。

和类的其他成员一样,被包容的类可以被声明为静态类,非静态被包容类称为内部类(inner class),也可以声明为抽象类或最终类,这些和一般类的声明是完全相同的。

下面我们通过两个例子说明内部类的使用。例 5.8 是用常规方法响应鼠标事件,例 5.9 是改写后的程序,使用了内部类。

**例 5.8** 在窗口中用鼠标拖动画线。运行结果如图 5.8 所示。

```

import java.applet. Applet;
import java.awt. * ;
import java.awt.event. * ;

```

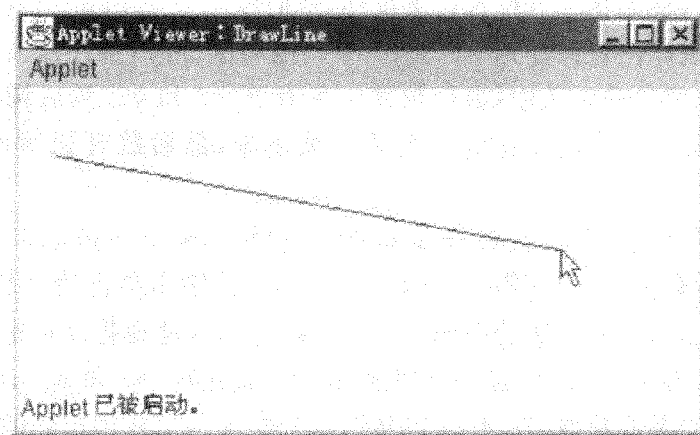


图 5.8

```

public class DrawLine extends Applet implements MouseListener, MouseMotionListener {
    int x1, y1, x2, y2;

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void paint(Graphics g) {
        g.drawLine(x1, y1, x2, y2);
    }

    public void mousePressed(MouseEvent e) { // 记录起点坐标
        x1=e.getX();
        y1=e.getY();
    }

    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mouseReleased(MouseEvent e) {}

    public void mouseDragged(MouseEvent e) { // 记录终点坐标
        x2=e.getX();
        y2=e.getY();
        repaint();
    }

    public void mouseMoved(MouseEvent e) {}
}

```

用鼠标画线时,首先在起点位置按下鼠标,此时发生一个 `mousePressed` 事件,在相应的处理方法中,将鼠标位置保存到 `x1` 和 `y1` 中。拖动鼠标时,将连续发生 `mouseDragged` 事件。在这个事件的处理方法中,鼠标位置被保存到 `x2` 和 `y2`,然后调用 `repaint` 方法重画直线。这样,鼠标不断移动,直线不断被重画出来,最后放开鼠标时,直线就固定下来了。

问题是,程序使用了 `MouseListener` 和 `MouseMotionListener` 接口,而我们必须实现这两个接口中的所有方法,不管用得到用不到它们。程序中是将这些用不到的方法以空方法的形式给出,即只给出方法的声明,不添加方法体,这使得程序中出现了多余的东西。

为了去掉这些多余的东西,我们可以利用 `MouseAdapter` 和 `MouseMotionAdapter`,它们是抽象类,称为事件裁剪器。你可以从这两个类继承那些用得到的事件处理方法,其他的可以不继承。这两个派生类将作为内部类添加到主类中,改写后的程序如下:

**例 5.9** 改写后的程序。运行结果参见图 5.8。

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class DrawLine1 extends Applet {
    int x1, y1, x2, y2;

    public void init() {
        addMouseListener(new MouseHandler());
        addMouseMotionListener(new MouseMotionHandler());
    }

    public void paint(Graphics g) {
        g.drawLine(x1, y1, x2, y2);
    }

    class MouseHandler extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            x1 = e.getX();
            y1 = e.getY();
        }
    }

    class MouseMotionHandler extends MouseMotionAdapter {
        public void mouseDragged(MouseEvent e) {
            x2 = e.getX();
            y2 = e.getY();
            repaint();
        }
    }
}
```

```

    }
}

```

例 5.9 在主类中添加了两个内部类 `MouseHandler` 和 `MouseMotionHandler`, 前者继承并修改了抽象类 `MouseListener` 中的 `mousePressed` 方法, 后者继承了 `MouseMotionAdapter` 中的 `mouseDragged` 方法并加以修改。

使用时, 直接将它们的构造方法作为 `addMouseListener` 或 `addMouseMotionListener` 方法的参数, 在参数传递中实现实例化。由于它们都被设计为内部类, 所以能自由访问主类的各个成员, 实现和例 5.8 完全相同的功能, 程序也更加简洁。如果不将它们声明为内部类, 就很难访问主类的非静态成员, 达不到同样的效果。

从这个例子可以看出, 使用内部类加强了类之间的联系, 为解决多继承问题又提供了一个有效的方法。下面再介绍一个匿名内部类的例子。所谓匿名类是指在声明一个类时不给它命名。

**例 5.10 用匿名类为窗口添加关闭事件处理方法。**

```

import java.awt.*;
import java.awt.event.*;

public class MyFrame {
    public static void main(String[] args) {
        new Frame1();
    }
}

class Frame1 extends Frame {
    Frame1() {
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setSize(350, 200);
        setVisible(true);
    }
}

```

在第 1 章里我们介绍过窗口应用程序, 遗憾的是, Java 的窗口只处理放大和缩小按钮事件, 不处理关闭按钮事件。要想关闭窗口, 必须自己编写关闭窗口事件的处理方法。

本例使用匿名类为窗口添加关闭事件的处理方法, 使程序非常简洁。从程序中可以看到, 匿名类的构造方法 `new WindowAdapter()` 直接作为 `addWindowListener` 方法的参数, 在括号内实现了类的继承和实例化, 并仅仅覆盖了 `WindowAdapter` 的 `windowClosing` 方法。

- 5-1 什么是类的继承性? 子类和父类有什么关系?
- 5-2 什么是类的多态性?
- 5-3 何为隐藏、覆盖、重载?
- 5-4 this 和 super 类有什么作用?
- 5-5 构造方法有何特点和作用?
- 5-6 什么是内部类、匿名类?
- 5-7 分析下面这段程序, 指出父类、子类以及它们的成员, 成员的作用是什么?

```
class Point {
    int x, y;
    Point(int a, int b) {setPoint(a,b);}
    public void setPoint(int a, int b) {x=a; y=b;}
}
class Circle extends Point {
    int radius;
    Circle(int a, int b, int r) {super(a,b); setRadius(r);}
    public void setRadius(int r) {radius=r;}
    public double area() {return 3.14159 * radius * radius;}
}
```

- 5-8 创建一个 Fraction 类执行分数运算。要求如下:
- (1) 用整型数表示类的 private 成员变量: f1 和 f2。
  - (2) 提供构造方法, 将分子存入 f1, 分母存入 f2。
  - (3) 提供两个分数相加的运算方法, 结果分别存入 f1 和 f2。
  - (4) 提供两个分数相减的运算方法, 结果分别存入 f1 和 f2。
  - (5) 提供两个分数相乘的运算方法, 结果分别存入 f1 和 f2。
  - (6) 提供两个分数相除的运算方法, 结果分别存入 f1 和 f2。
  - (7) 以 a/b 的形式打印 Fraction 数。
  - (8) 以浮点数的形式打印 Fraction 数。
  - (9) 编写主控程序运行分数运算。

5-9 根据你能想到的二维、三维图形设计一个树状层次结构。以 Shape 为父类, 从中派生出 D2Shape 和 D3Shape, 定义它们的所有类。

# 第6章

## 包、接口和异常

本章介绍 Java 语言程序包(package)、接口(interface)和异常处理(exception)的使用方法。

程序包是类和接口的集合。利用程序包可以把你常用的类或功能相似的类放在一个程序包中。Java 语言还提供了系统程序包,其中包含了大量的类,可以在编写 Java 程序时直接引用它们。

接口解决了 Java 不支持多重继承的问题,可以通过实现多个接口达到与多重继承相同的功能。

处理程序运行时的错误和设计程序同样重要,只有能够完善处理运行时出错的程序才能长期稳定地运行,异常处理就是用来处理程序运行时出错的。下面分别进行介绍。

### 6.1 程序包

在 Java 语言中,程序包的概念和其他编程语言中函数库的概念是相同的,也可称为类库。在 Java 系统中已经包含了设计者编写的大量系统程序包,我们除了了解如何使用系统程序包外,还要学习如何把自己编写的类组成程序包的形式,以便将来像使用系统程序包一样使用自己的程序包。

#### 6.1.1 声明自己的程序包

要使用程序包,首先要声明。程序包声明的方式为:

```
package 程序包名
```

程序包声明必须被加到源程序文件的起始部分,表示该文件的全部类都属于这个程序包。你还可以在不同的文件中使用相同的程序包声明语句,这样就可将不同文件中的类都包含在相同的程序包中了。下面我们举例说明如何建立自己的程序包。

**例 6.1** 在文件 bl.java 中,包含有如下语句:

```
package Mypackage
```

```
class MyClass1 {
    ...
}
```

**例 6.2** 在文件 b2.java 中, 包含有如下语句:

```
package Mypackage
class MyClass2 {
    ...
}
class MyClass3 {
    ...
}
```

通过这两个程序文件即可得到自己声明的程序包 Mypackage, 程序包中包含有类 MyClass1、MyClass2 和 MyClass3。

注意: 程序包声明语句“package 程序包名称”必须放在程序文件的第一行, 而且前面不能有注释和空格。

### 6.1.2 程序包的引用

在 Java 程序中怎样告诉编译器, 使用哪些程序包中的类呢? 使用程序包中的类时, 一般在 Java 程序的开头使用 import 语句指明含有该类的程序包。如下面的语句:

```
import java.awt.Graphics;
import java.applet.*;
```

“java.awt.Graphics”表示程序要使用 Java 的 awt 系统程序包中的 Graphics 类, 编译器就会从 awt 程序包中查找类 Graphics。

“java.applet.\*”表示程序中需要引入 applet 程序包中的全部类。

### 6.1.3 Java 的系统程序包

Java 提供了大量的类, 为便于管理和使用, 分为若干个程序包。程序包又称类库或 API 包, 所谓 API(Application Program Interface)即应用程序接口。API 包一方面提供丰富的类与方法供我们使用, 如画图形、播放声音等, 另一方面又负责和系统软硬件打交道, 把用户程序的功能圆满实现。

所有 Java API 包都以“java.”开头, 以区别用户创建的包。表 6.1 列出了常用的程序包和所支持的功能。

#### 1. java.lang 程序包

是 Java 语言的基础类库, 包含基本数据类型、数学函数、字符串类等。这是唯一自动引入每个 Java 程序的类库。

下面是 java.lang 程序包中包含的主要类:



表 6.1 Java 常用类库

. API 包	功 能
java. lang	包含 Java 语言的核心类库
java. util	提供各种实用工具类
java. io	标准输入/输出类
java. net	实现 Java 网络功能的类库
java. awt	组建标准 GUI, 包含了众多的图形组件、方法和事件
java. applet	提供对通用 Applet 的支持, 是所有 Applet 的基类
java. security	支持 Java 程序安全性

- (1) 数据类型类 BigDecimal、BigInteger、Byte、Double、Float、Integer、Long、Short;
- (2) 基本数学函数 Math 类;
- (3) 用于字符串处理的 String 类和 StringBuffer 类;
- (4) System、Object 类;
- (5) 线程 Thread 和 ThreadDeath 类。

## 2. java. util 程序包

包含一些低级的实用工具类。这些实用工具类使用方便, 而且很重要。主要有: 日期 Date 类、堆栈 Stack 类、随机数 Random 类、向量 Vector 类等。

## 3. java. io 程序包

是 Java 语言的输入输出类库, Java 语言的文件操作都是由该类库中的输入输出类来实现的。此外该类库还提供了一些与其他外部设备交换信息的类。java. io 程序包除了包含标准输入、输出类外, 还有缓存流、过滤流、管道流和字符串类等。

## 4. java. net 程序包

含有访问网上资源的 URL 类, 用于通信的 Socket 类和网络协议子类库等。Java 语言是一门适合分布式计算环境的程序设计语言, 网络类库正是为此设计的。其核心就是对 Internet 协议的支持, 目前该类库支持多种 Internet 协议, 包括 HTTP、Telnet、FTP 等等。

## 5. java. awt 程序包

提供了创建图形用户界面的全部工具。它包括许多我们熟悉的图形组件(component)类, 如窗口、对话框、按钮、复选框、列表、菜单、滚动条和文本区等类; 用于管理组件排列的布局管理器 Layout 类; 以及常用的颜色 Color 类、字体 Font 类。java. awt. event 类库用来处理各种不同类型的事件。

## 6. java.applet 程序包

java.applet 是所有小应用程序的基础类库。它只包含了一个 Applet 类,所有小应用程序都是从该类中派生的。

## 7. java.security 程序包

包括 java.security.acl 和 java.security.interfaces 子类库,利用这些类可对 Java 程序进行加密,设定相应的安全权限等。

## 6.2 接口

什么是接口?所谓接口可以看作是方法和常量的一个集合。接口与抽象类相似,接口中的方法只是做了声明,而没有定义任何具体操作。使用接口是为了解决 Java 语言中不支持多重继承的问题。Java 不支持多重继承是为了使语言本身更加简单,但同时也就限制了语言的功能。为了平衡这种情况,Java 使用接口来弥补多重继承的功能。

如何使用接口呢?可在类的声明中使用关键字 implements,声明该类实现了某个或多个接口,同时在类中实现所有接口中定义的方法。注意:Java 程序中的类获得接口中声明的一个方法不能视为继承,因为接口只是提出了某种功能的规范表达而没有具体实现。所以,类引用接口不叫继承而称为实现。

### 6.2.1 声明接口

要使用接口,首先要声明它,如同使用对象一样。声明接口的语法格式如下:

```
[修饰符] interface 接口名 {  
    ... // 静态常量及方法声明  
}
```

**例 6.3** 在下面的程序中声明了一个接口。

```
public interface Range {  
    int MIN=0;  
    int getMaxRange();  
    void setMaxRange(int value);  
}
```

说明:在本例中定义了一个接口 Range,在这个接口中声明了两个方法: getMaxRange()和 setMaxRange(int) 以及一个整型变量 MIN。

注意:Java 编译器和解释器自动把接口中声明的变量当作 public static final 类型的变量(即常量),不管是否使用了这些修饰符,所以接口中的变量不能被修改。接口中声明的变量都必须被初始化,否则会产生编译错误。

接口中的方法默认为 `public abstract`, 不管有没有这些修饰符。与类、成员变量及方法一样, 接口也可以使用一些访问控制修饰符进行限制, 当然也可以不用。如果使用了 `public`, 那么表示该接口可被任意的类实现。如果没有使用 `public` 修饰符, 那么就只有与接口在同一个程序包中的类才可以实现这个接口。

### 6.2.2 接口的继承关系

在 Java 中, 和类一样, 接口可以实现继承, 一个接口可以继承父接口的所有成员。

#### 1. 接口的单继承

我们用下面的例子来说明接口的单继承。

**例 6.4** 一个具有单继承的接口。

```
interface A {  
    void F1();  
}  
  
interface B extends A {  
    void F2();  
}
```

这里声明的接口 B 继承了接口 A 中的所有数据和方法, 这种接口之间的继承称为单继承。接口之间的继承与类的继承一样使用关键字 `extends`。

如果一个类实现了接口 B, 那么它必须实现接口 A 和接口 B 中的全部方法, 看下面的例子:

**例 6.5** 在类中引用具有继承接口的方法。

```
class MyClass implements B {  
    void F1() {  
        ...  
    }  
    void F2() {  
        ...  
    }  
}
```

说明: 在这个例子中 `MyClass` 实现了接口 B, 因为接口 B 继承了接口 A, 所以接口 B 实际上还包含了接口 A 中的方法 `F1`。因此在类 `MyClass` 中必须实现接口 A 中的方法 `F1` 和接口 B 中的方法 `F2`。

#### 2. 接口的多重继承

在 Java 语言中, 不支持类的多重继承。但是支持接口的多重继承, 其语法格式如下:

interface 接口名 extends 接口名 1, 接口名 2, ...

可见接口的多重继承只是在单继承的基础上再加上几个接口,并把这些接口用逗号分隔开。

### 6.2.3 在类中实现一个接口

为了在类中实现一个接口,必须用 implements 关键字和接口名声明一个类。下面我们例子介绍怎样实现在例 6.3 中定义的接口 Range。

**例 6.6** 在类中实现一个接口的方法。

```
class MyClass1 implements Range {  
    ...  
    int getMaxRange() {  
        ...  
    }  
    void setMaxRange(int value) {  
        ...  
    }  
}
```

说明:在声明类 MyClass1 的语句中使用“implements Range”引用了接口 Range,并在类体中实现了接口的方法 getMaxRange()和 setMaxRange(int value)。

### 6.2.4 在类中实现多个接口

Java 语言中规定,类之间只能是单继承,但可以实现多个接口。看下面的例子是如何在类中实现多个接口的。

**例 6.7** 在类中实现多个接口的方法。

```
class MyClass2 implements Range, Range1 {  
    ...  
    int getMaxRange() {  
        ...  
    }  
    void setMaxRange(int value) {  
        ...  
    }  
    ... // Range1 的方法实现  
}
```

说明:在例子中,通过“implements Range, Range1”实现了两个接口。Range 是例 6.3 定义的接口,Range1 是另外一个接口。

注意:引用接口时,必须实现接口中的所有方法。由于接口中的方法被声明为抽象方法,根据第 4 章的介绍,一个抽象方法不能出现在非抽象类中,因此,你创建的类如果不是

抽象类就必须实现接口的所有方法。

## 6.3 异常处理

本节介绍 Java 的异常处理机制。异常是用来处理程序错误的有效机制,以往需要由程序员完成的程序出错情况判别,在 Java 中改为由系统承担。通过系统抛出的异常,程序可以很容易地捕获并处理发生的异常情况。

### 6.3.1 什么是异常

在程序执行期间,会有许多意外的事件发生。例如,程序申请内存时没有申请到、对象还未创建就被使用等。这些异常情况的处理对于简单程序或个人使用的程序来说,可能并不重要,因为这些问题如果一个一个地都要处理会占用我们大量的精力,甚至比我们考虑程序的功能还要麻烦。

对于一个应用软件,异常处理是不可缺少的。你必须在程序中考虑每一个可能发生的异常情况并进行处理,以保证程序在任何情况下都能正常运行。事实证明,一个仔细设计了异常处理的程序,可以长时间的可靠运行,而不容易发生致命的错误,如程序被迫关闭、甚至系统终止等等。所以学会进行异常情况处理对于想编写功能完善且具有实用价值的程序员来说是必不可少的。

在了解异常处理之前,我们先来了解什么是异常。当 Java 程序违反了 Java 语言的语义限制时,Java 虚拟机就会发出出错信号,这就是异常。例如,被 0 除、数组下标越界就是一种违规行为。某些程序语言在处理这种异常时会强制终止程序,而另一些程序语言可能允许以一种不确定的方式运行程序。这两种处理方式和 Java 语言的健壮性目标都不相符。

Java 是这样规定的:当语义限制被违反时,将会抛出异常,并将引起程序流程从异常发生点转移到程序员指定的位置。异常一般是从发生的地方被抛出,在流程控制转移到的地方被捕获。

Java 程序也可以使用 throw 语句主动抛出异常。如果你使用过 C 语言编程可能还会记得,让一个函数返回 0 或 -1 来表示函数是否被正确地执行。这种利用某些特殊的数值表示异常情况的方法,经常会被函数调用者忽视,从而导致了程序的非健壮性。

Java 中的每一种异常都是类 Throwable 或其子类的对象,这种对象可以用来把异常发生的信息传送给捕获它的处理程序。处理程序是由 try...catch 语句建立的。

在抛出异常的过程中,Java 虚拟机一个接一个地终止当前线程中的表达式、语句、方法和构造方法的调用,以及已经开始但还未执行完的静态初始化程序等。这个过程要一直持续到找到异常处理程序为止。如果找不到处理程序,那么方法 unCaughtException 被当前进程的父进程 ThreadGroup 调用。

### 6.3.2 异常发生的原因

异常发生的原因有以下三种:

(1) Java 虚拟机检测到了非正常的执行状态,这些状态可能是由以下几种情况引起的:

- ① 表达式的计算违反了 Java 语言的语义,例如整数被 0 除;
- ② 在载入或链接 Java 程序时出错;
- ③ 超出了某些资源限制,例如使用了太多的内存。

这些异常都是无法预知的。

(2) Java 程序代码中的 throw 语句被执行。

(3) 异步异常发生。异步异常的原因可能有:

- ① Thread 的 stop 方法被调用;
- ② Java 虚拟机内部错误发生。

### 6.3.3 编译时对异常情况的检查

#### 1. 可检测的异常

在编译时,编译器分析哪些方法会产生可检测的异常,然后检查方法中的可检测异常的处理部分。如果方法中没有异常处理部分,就要在方法的 throws 子句说明该方法会抛出但不捕获的异常,以告知调用它的其他方法,即将异常上交给调用者处理。

例如,类 Thread 的方法 sleep 定义如下:

```
public static void sleep (long millis) throws InterruptedException {...}
```

说明 sleep 方法会抛出 InterruptedException 类型的异常,也就是说,这种异常将不再被 sleep 捕获(方法中没有 try...catch),此时可顺利通过编译。否则,必须在 sleep 方法中加入 try...catch 来处理这种异常才能通过编译。

#### 2. 不可检测的异常

不可检测的异常类是 RuntimeException 及其子类、Error 及其子类,其他的异常类则是可检测的类。标准 Java API 定义了许多异常类,既包括可检测的,也包括不可检测的。由程序员定义的异常类也可以包含可检测类和不可检测类。

编译器对不可检测的异常类不进行检查。为什么不检查它们?这是因为 Java 的设计者认为检测这些异常对 Java 程序的正确性方面帮助不大,而且这种情况发生的原因很多,例如整数除以 0、数组越界等。如果对每一种情况都进行处理会很麻烦,所以这些异常在编译时不检查。编译器也不检测 Error 异常。因为这种错误可能发生在程序中的许多地方,并且这种异常恢复起来很困难或者根本不可能恢复,所以检测这类错误是不必要的。

### 6.3.4 异常的层次结构

Java 程序中的异常以类的层次结构组织。Throwable 是所有异常类的父类,它是 Object 的直接子类。Exception 和 Error 是 Throwable 的直接子类,而 RuntimeException 是 Exception 的子类,如图 6.1 所示。

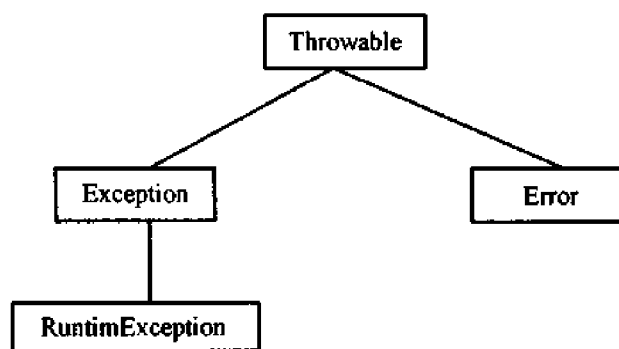


图 6.1

### 6.3.5 Java 定义的标准异常类

在系统包 `java.lang`、`java.util`、`java.io`、`java.net` 中声明的异常类是标准异常类。这些标准异常类分为两种：一种是 `RuntimeException` 子类，因为 `RuntimeException` 是不可检测的异常类，所以这些标准异常类也是不可检测的异常类；另一种是除了 `RuntimeException` 子类以外的其他 `Exception` 的子类，这些异常类是可检测的。

下面我们分别介绍这两种标准的异常类。

#### 1. 不可检测的标准异常类

##### (1) `java.lang` 中的标准异常类

- `ArithmeticException` 表示遇到了异常的算术问题，例如被 0 整除。
- `ArrayStoreException` 试图把与数组类型不相符的值存入数组。
- `ClassCastException` 试图把一个对象的引用强制转换为不合适的类型。
- `IndexOutOfBoundsException` 下标越界。
- `NullPointerException` 试图使用一个空的对象引用。
- `SecurityException` 检测到了违反安全的行为。

##### (2) `java.util` 中的标准异常类

- `EmptyStackException` 试图访问一个空堆栈中的元素。
- `NoSuchElementException` 试图访问一个空向量中的元素。

#### 2. 可检测的标准异常类

##### (1) `java.lang`

- `ClassNotFoundException` 具有指定名字类或接口没有被发现。
- `CloneNotSupportedException` 克隆一个没有实现 `Cloneable` 接口的类。
- `IllegalAccessException` 试图用给出了完整的路径信息的字符串加载一个类，但是当前正在执行的方法无法访问指定类，因为该类不是 `public` 类型或在另一个包中。
- `InstantiationException` 试图使用 `Class` 的 `newInstance` 方法创建一个对象实例，但指定的对象没有被实例化，因为它是一个接口、抽象类或者一个数组。

- `InterruptedException` 当前的线程正在等待,而另一个线程使用了 `Thread` 的 `interrupt` 方法中断了当前线程。

(2) `java.io`

- `IOException` 申请 I/O 操作没有正常完成。
- `EOFException` 在输入操作正常结束前遇到了文件结束符。
- `FileNotFoundException` 在文件系统中,没有找到由文件名字符串指定的文件。
- `InterruptedIOException` 当前线程正在等待 I/O 操作完成,而另一个线程使用 `Thread` 的 `interrupt` 方法中断了当前线程。

### 6.3.6 异常的处理

当一个异常抛出(即产生了异常)时,该如何处理呢?

在 Java 语言中使用语句 `try...catch...finally` 进行异常处理,程序流程从引起异常的代码转移到最近的 `try` 语句的 `catch` 子句,如下例所示。

**例 6.8** 异常处理语句结构。

```
try {...} // 被监视的代码段,一旦发生异常,则交由其后的 catch 代码段处理
catch (异常类型 e) {...} // 要处理的第一种异常
catch (异常类型 e) {...} // 要处理的第二种异常
...
finally {...} // 最终处理
```

说明:我们把可能会发生异常情况的代码放在 `try` 语句段中,利用 `try` 语句对这组代码进行监视。如果发生了第一种异常,使用第一个 `catch` 中的代码段处理;如果发生了第二种异常,则使用第二个 `catch` 中的代码段处理。

`catch` 语句在执行前,必须识别抛出的异常是 `catch` 能够捕获的异常。如果 `catch` 语句参数中声明的异常类与抛出的异常类相同,或者是它的父类,`catch` 语句就可以捕获任何这种异常类的对象。

如果发生的异常没有捕获到,会发生什么样的情况呢?那么流程控制将沿着调用堆栈一直向上传。也就是说,发生异常的方法 1 如果没有处理异常的话,就把异常传给调用它的方法 2,如果方法 2 也没有处理异常,那么就把异常再传给调用它的方法 3,这样把异常一直传到能够处理它的方法。

如果直到最后还是没有发现处理异常的 `catch` 语句,那么在 `finally` 子句执行完后,调用 `ThreadGroup` 的 `unCaughtException` 方法,终止当前的线程(即发生了异常的线程)。

如果你希望在异常发生时能确保有一段代码被执行,那么应该使用 `finally` 子句。这样即使发生的异常与 `catch` 所能捕获的异常不匹配也会执行 `finally` 子句。看下面的例子:

**例 6.9** `try...catch` 语句处理异常的过程。运行结果如图 6.2 所示。

```
public class CatchDemo1 {
    public static void main(String[] arg3) {
```



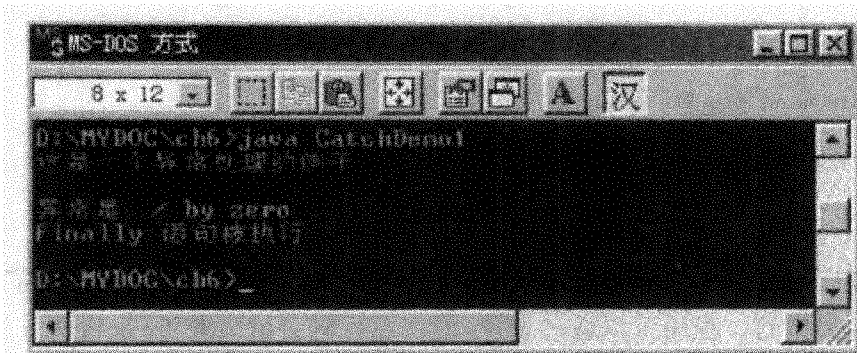


图 6.2

```

System.out.println("这是一个异常处理的例子\n");
try {
    int i=10;
    i /= 0;
}
catch (ArithmeticException e) {
    System.out.println("异常是:" + e.getMessage());
}
finally {
    System.out.println("finally 语句被执行");
}
}
}

```

这是一个简单的异常处理。在程序中主动产生一个被 0 除的异常，然后用 catch 语句捕获这种异常。因为被 0 除的异常是一种 ArithmeticException 类的异常，所以 catch 语句可以捕获它并作出相应的处理。

如果在 catch 语句中我们声明的异常类是 Exception，catch 语句也能正确地捕获，这是因为 Exception 是 ArithmeticException 的父类。如果你不能确定会发生哪种情况的异常，那么你最好指定 catch 的参数为 Exception。否则，如果试图捕获一个不同类型的异常，将会发生意想不到的情况。我们看例 6.10。

**例 6.10** catch 语句中声明的异常类型不匹配的情况。

```

public class CatchDemo2 {
    public static void main(String[] args) {
        System.out.println("这是一个异常处理的例子\n");
        try {
            int i=10;
            i /= 0;
        }
        catch (IndexOutOfBoundsException e) {
            System.out.println("异常是:" + e.getMessage());
        }
    }
}

```

```

    }
    finally {
        System.out.println("finally 语句被执行");
    }
}

```

上面的程序试图捕获一个异常类为 `IndexOutOfBoundsException` 的异常,但发生的异常却是其他类型的。程序可以通过编译,但在运行时会给出警告:报告所发生的但没有被捕获的异常。不过在此之前,finally 语句将会被执行。

在某些情况下,同一段程序可能产生不止一种异常情况。你可以放置多个 catch 子句来检查每一种异常类型,第一个与异常匹配的 catch 就会被执行。如果一个异常类和其子类都出现在 catch 子句中,应把子类放在前面,否则将永远不会到达子类。下面是有两个 catch 子句的程序例子。

**例 6.11** 多个 catch 子句的异常处理。运行结果如图 6.3 所示。

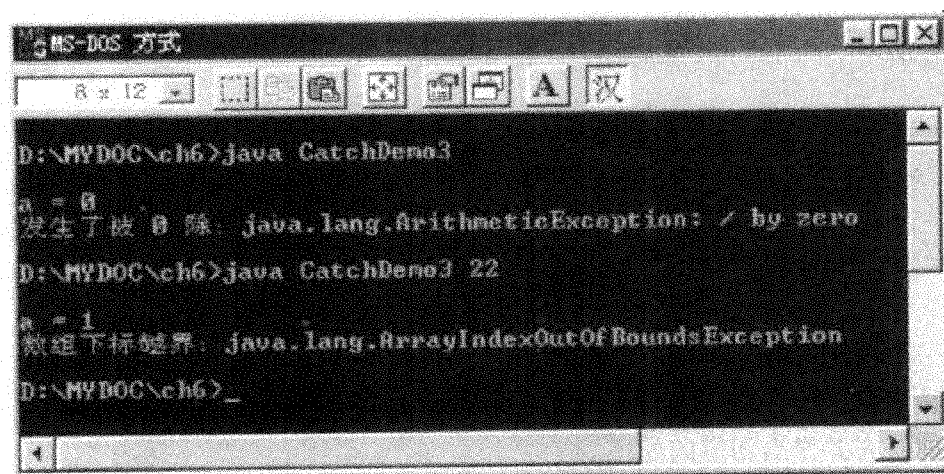


图 6.3

```

public class CatchDemo3 {
    public static void main(String[] args) {
        try {
            int a=args.length;
            System.out.println("\na = "+a);
            int b=42/a;
            int c[]={1};
            c[42]=99;
        }
        catch (ArithmeticException e) {
            System.out.println("发生了被 0 除:"+e);
        }
        catch (ArrayIndexOutOfBoundsException e) {

```

```

        System.out.println("数组下标越界:" + e);
    }
}
}

```

如果在程序运行时不输入参数, args.length 的值为 0, 这将引起一个被 0 除异常。如果我们提供一个命令行参数, 如 22, 将不会引起被 0 除异常, 但会引起另一个异常, 即数组下标越界 `ArrayIndexOutOfBoundsException`。因为整型数组只有一个元素 `c[0]`, 程序中却要使用 `c[42]`, 这将发生数组下标越界。

### 6.3.7 创建自己的异常

前面讲述了 Java 中的系统异常处理机制, 现在我们介绍怎样创建自定义的异常。通常自定义异常是从 `Exception` 类中派生的, 可使用下面的声明语句来创建。

```
class 自定义异常名 extends Exception {...}
```

**例 6.12** 创建自定义异常。运行结果如图 6.4 所示。

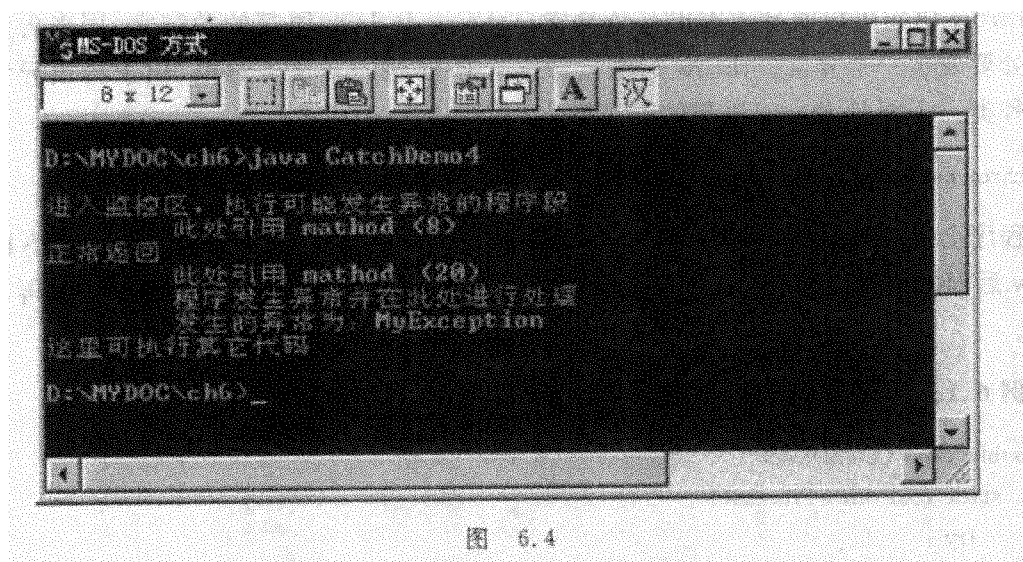


图 6.4

```

class MyException extends Exception {
    private int x;
    MyException(int a) {x=a;}
    public String toString() {return "MyException";}
}

public class CatchDemo4 {
    static void method(int a) throws MyException { // 声明方法会抛出 MyException
        System.out.println("\t 此处引用 method (" + a + ")");
        if (a > 10) throw new MyException(a); // 主动抛出 MyException
        System.out.println("正常返回");
    }
}

```

```

public static void main(String args[]) {
    try {
        System.out.println("\n 进入监控区, 执行可能发生异常的程序段");
        method(8);
        method(20);
        method(6);
    }
    catch (MyException e) {
        System.out.println("\t 程序发生异常并在此处进行处理");
        System.out.println("\t 发生的异常为:" + e.toString());
    }
    System.out.println("这里可执行其他代码");
}
}

```

### 6.3.8 throw 语句

throw 语句可以明确地抛出一个异常。throw 是 Java 语言的关键字, 用来告知编译器此处要发生一个异常。throw 后面要跟一个新创建的异常类对象, 用于指出异常的名称和种类。下面是 throw 语句的语法形式:

```
throw new someException();
```

程序会在 throw 语句处立即终止, 转向 try...catch 寻找异常处理方法, 不再执行 throw 后面的语句。在例 6.12 中就使用了 throw 语句主动抛出一个异常。下面再看一个例子。

**例 6.13** throw 语句的使用。运行结果如图 6.5 所示。

```

public class CatchDemo5 {
    static void throwProcess() {
        try {
            throw new NullPointerException("空指针异常");
        }
        catch (NullPointerException e) {
            System.out.println("\n 在 throwProcess 方法中捕获一个空指针异常");
            throw e;
        }
    }

    public static void main(String args[]) {
        try {
            throwProcess();
        }
        catch (NullPointerException e) {

```

```

        System.out.println("再次捕获:" + e);
    }
}

```



图 6.5

主程序首先调用 `throwProcess` 方法,明确地抛出了一个 `NullPointerException` 异常并将其命名为“空指针异常”。然后程序流程将转向 `throwProcess` 方法中的 `catch` 子句,输出一条信息。注意,`throwProcess` 方法的 `catch` 子句中紧接着又抛出了一个同样的异常 `e`,这时程序流程将返回到主程序,由 `catch` 子句捕获这个异常。

### 6.3.9 throws 语句

`throws` 用来表明一个方法可能抛出的各种异常。对大多数 `Exception` 子类和自定义异常类来说,Java 编译器会强迫你在一个方法的声明语句中表明会抛出的异常类型。如果异常的类型是 `Error`、`RuntimeException` 或它们的子类,这个规则不起作用,因为这些异常在程序编译期间不会出现。如果你想明确抛出一个 `RuntimeException` 或自定义异常类,就必须在方法的声明语句中用 `throws` 子句来表明它的类型,以便通知调用这个方法的其他方法准备捕获它。`throws` 子句必须位于大括号之前,以下是 `throws` 的语法形式:

返回值类型 方法名(参数) throws 异常类型 {}

下面这段代码,抛出了一个异常,但既没有捕获它,也没有用 `throws` 来声明。这在编译时将不会通过。

```

class ThrowsDemo1 {
    static void method() {
        System.out.println("在 method 中抛出一个异常");
        throw new IllegalAccessException();
    }

    public static void main(String args[]) {
        method();
    }
}

```

```

    }
}

```

为了让这段代码编译过去,需要声明 `method` 方法会抛出 `IllegalAccessException`,并且不在 `method` 中捕获它,而在调用 `method` 的 `main` 方法里捕获它。看下面正确的例子:

**例 6.14** `throws` 语句的使用。运行结果如图 6.6 所示。

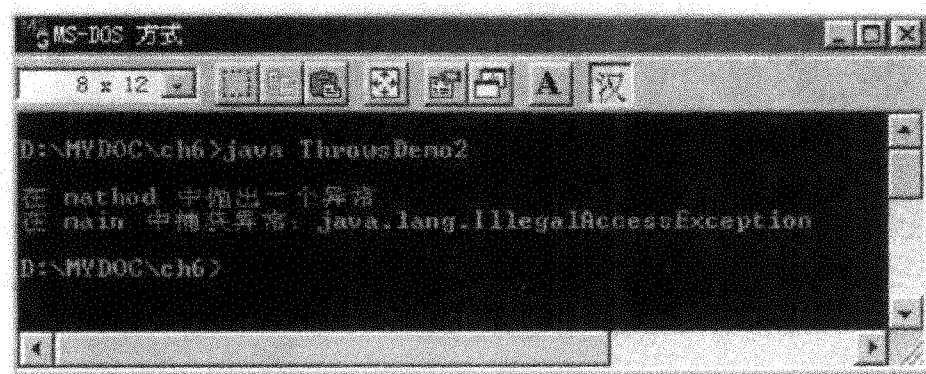


图 6.6

```

class ThrowsDemo2 {
    static void method() throws IllegalAccessException {
        System.out.println("\n 在 method 中抛出一个异常");
        throw new IllegalAccessException();
    }

    public static void main(String args[]) {
        try {
            method();
        }
        catch (IllegalAccessException e) {
            System.out.println("在 main 中捕获异常:" + e);
        }
    }
}

```

如果在 `method` 方法中加入异常处理程序也能通过编译并正确运行。下面是改写后的程序,在 `method` 中抛出异常并在 `method` 中捕获这个异常。

```

class ThrowsDemo3 {
    static void method() {
        try {
            System.out.println("\n 在 method 中抛出一个异常");
            throw new IllegalAccessException();
        }
        catch (IllegalAccessException e) {

```

```

        System.out.println("在 method 中捕获异常:" + e);
    }
}
public static void main(String args[]) {
    method();
}
}

```

### 6.3.10 finally 语句

当一个异常被抛出时,程序的执行就不再是连续的了,会跳过某些语句,甚至会由于没有与之匹配的 catch 子句而过早地返回。有时要确保一段代码不管发生什么异常都能被执行到是必要的,关键字 finally 就是用来标识这样一段代码的。即使没有 catch 子句,finally 语句块也会在执行了 try 语句块后立即被执行。每个 try 语句至少都要有一个与之相配的 catch 或 finally 子句。

一个方法要返回到调用它的方法,或者是通过 return 语句,或者是通过一个没有被捕获的异常,但 finally 子句总是在方法返回前执行。

**例 6.15** finally 子句的使用。运行结果如图 6.7 所示。

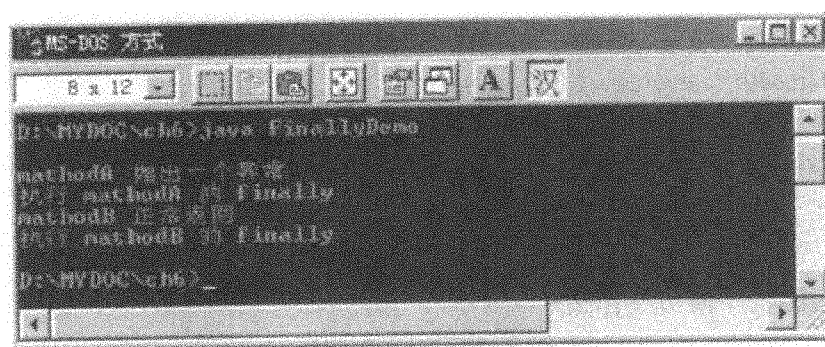


图 6.7

```

class FinallyDemo {
    static void methodA() {
        try {
            System.out.println("\nmethodA 抛出一个异常");
            throw new RuntimeException();
        } finally {System.out.println("执行 methodA 的 finally");}
    }
    static void methodB() {
        try {
            System.out.println("methodB 正常返回");
            return;
        } finally {System.out.println("执行 methodB 的 finally");}
    }
    public static void main(String args[]) {

```

```

    try {
        methodA();
    } catch (Exception e) {methodB();}
}
}

```

在这个例子中有两个方法,分别用不同的途径退出,但都执行了 finally 子句。

## 习 题

6-1 程序包有什么作用? 有哪些类型的程序包? 创建一个自己的程序包,其中应包含 4 个类。

6-2 接口有什么作用? 创建一个接口并在类中实现这个接口。

6-3 什么是异常? 为什么要进行异常处理?

6-4 如何创建一个自定义异常?

6-5 如何抛出系统异常? 如何抛出自定义异常?

6-6 下面的程序有何错误?

```

public class Quiz1 {
    public static void main(String args[]) {myMethod();}
    myMethod() {throw new MyException();}
}
class MyException {
    public String toString() {return "自定义异常";}
}

```

6-7 下面的程序输出是什么?

```

public class Quiz2 {
    public static void main(String args[]) {
        try {throw new MyException();}
        catch (Exception e) {System.out.println("It's caught!");}
        finally {System.out.println("It's finally caught!");}
    }
}
class MyException extends Exception{}

```

选择答案:

- ① It's finally caught!
- ② It's caught!
- ③ It's caught!  
It's finally caught!
- ④ 无输出



# 第7章

## 常用系统类

本章介绍 Java 常用的系统类,包括 Java Applet、字符串类、标准输入/输出、数学函数类、日期类、随机数类以及向量类等。在 Java 程序设计中,这些类起着重要的作用,它们是 Java 设计者已经编写好的程序代码,程序员可以在程序中直接引用。

### 7.1 Applet 类

Applet 是一种特殊的 Java 程序,经过编译后可被嵌入到 HTML 文件中,并由 Web 浏览器内置的 Java 解释器执行。所有 Applet 都继承自同一个类:java.applet.Applet,这个类是 Java 语言的基础类,有关 Applet 的所有特性都被定义在这个类中。

#### 7.1.1 Applet 简介

Applet 是一种在 Web 页中运行的小应用程序,广泛用于创建动态的、交互式的 Web 应用程序。Java 的 Applet 与 Application 有一个明显的区别:Application 是独立程序,可直接通过 Java 解释器来解释运行;而 Applet 只能在 Web 浏览器上运行,必须通过 <APPLET> 标签将编译后的字节码文件嵌入到 Web 页中。

当用户加载一个带有 Applet 的 Web 页时,浏览器将从 Web 服务器上下载 Applet,并在本地计算机上运行此 Applet。出于安全上的考虑,Java 设计者对 Applet 做了一些必要的限制。否则,Applet 可能会对网络造成破坏,或突破系统的安全防护。这些限制有以下几个方面:

- (1) 不能加载库函数或定义本地方法;
- (2) 不能读写本地计算机上的文件系统;
- (3) 除了下载它的服务器外不能和其他主机建立网络通信;
- (4) 不能运行本地计算机上的任何程序;
- (5) 不能读取某些系统特性;
- (6) Applet 打开的窗口和 Application 打开的窗口在外观上有所不同。

除此之外,Java 语言本身还在编译器及解释程序中包含了各种形式的安全性和一致性检查,以防止语言的错误运用。这种限制与安全性检查的结合,使得有恶意的 Applet

很难对用户的系统造成破坏。

另一方面,java.applet 包提供了 Application 所不具备的一些 API 接口,例如 Applet 可以播放声音而其他程序则不能。下面是 Applet 能够做的事情:

- (1) 可以和它所在的 Web 服务器建立网络连接;
- (2) 能使 Web 浏览器方便地显示 HTML 文档内容;
- (3) 离开网页后 Applet 可以继续运行也可以停止运行;
- (4) 可在状态栏显示短信息;
- (5) 可以调用同一个网页上的其他 Applet 中的公共方法;
- (6) 可以播放声音文件;
- (7) 可以从 HTML 的<APPLET>标签中获取参数;
- (8) 从本地计算机上加载 Applet 时没有从网络加载时所附带的限制。

### 7.1.2 Applet 的生命周期

下面是一个简单的 Java Applet,它揭示了 Applet 的主要活动。

**例 7.1** 一个显示生命周期的 Applet,如图 7.1 和图 7.2 所示。

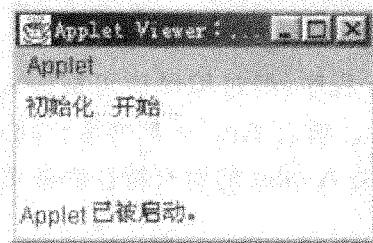


图 7.1

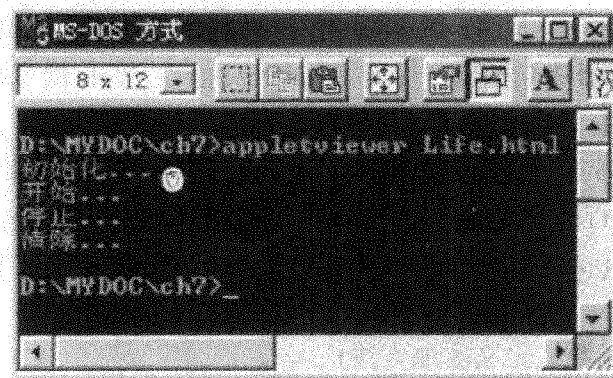


图 7.2

```
import java.applet, Applet;  
import java.awt, Graphics;  
  
public class Life extends Applet {  
    StringBuffer buffer=new StringBuffer();
```

```

public void init() {
    addWords("初始化...");
}

public void start() {
    addWords("开始...");
}

public void stop() {
    addWords("停止...");
}

public void destroy() {
    addWords("清除...");
}

void addWords(String s) {
    System.out.println(s);
    buffer.append(s);
    repaint();
}

public void paint(Graphics g) {
    g.drawString(buffer.toString(), 5, 15);
}
}

```

上例揭示了一个 Applet 的生命周期, 4 个方法对应着 4 个阶段: init(初始化)、start(开始运行)、stop(停止运行)、destroy(清除)。每个方法都调用自定义方法 addWords 来显示相应的字符串。addWords 方法首先在标准输出即屏幕上显示字符串参数, 然后将字符串添加到字符串缓冲区 buffer, 最后调用 repaint 方法重画, 而 repaint 方法则自动调用 paint 方法在指定位置显示字符串。g 是 Graphics 对象, 代表着一个图形界面积即显示区域, 它继承了 Graphics 的各种画出方法。注意: buffer 是 StringBuffer 类型, 可调用 toString 方法转换成 String 类型输出。

从图 7.1 中看到 Applet 显示的内容为“初始化... 开始...”, 表明当 Applet 出现时, 首先执行了 init, 然后是 start。当关闭它时, 先执行 stop 再执行 destroy。你可以从图 7.2 看到这个 Applet 的整个生命周期。下面, 我们概括出它的主要活动内容:

### 1. 初始化

当 Applet 首次加载(或重新加载)时, 要进行初始化操作。初始化可进行建立添加到程序中的对象、设置对象初始状态、为成员变量赋初值、加载图像或字体等操作, 这些可通过覆盖 init 方法来实现。

## 2. 开始运行

Applet 在初始化完成后即被调用,如果运行已被停止,还可以再次启动。例如,当用户链接到了另外一个网页,原网页嵌入的 Applet 即停止运行(如程序中有 stop);当用户再次返回到原网页时,该 Applet 将重新开始运行。尽管在一个生命周期内,Applet 可以多次开始运行,但初始化只能进行一次。

你可以在 start 方法中实现 Applet 的功能,也可以启动一个或多个线程来完成任务。一般情况下,如果覆盖了 start 方法就要覆盖 stop 方法。

## 3. 停止运行

当用户离开含有正在运行 Applet 的网页时,浏览器将自动调用 stop 方法。如果 stop 方法中含有尚未结束的任何线程,这些线程就会进入休眠,直到 Applet 被重新启动。线程在休眠时不占用系统资源。如果没有 stop 方法,Applet 会一直运行。

## 4. 清除

很多 Applet 不需要使用 destroy 方法,因为 stop 方法(在 destroy 之前被调用)已经把关闭 Applet 所必需的工作完成了。另外,关闭浏览器时,Java 解释器会自动调用默认的清除方法,以释放 Applet 占用的系统资源。前几章介绍的例子都没有使用 destroy 方法。

## 5. 画出

paint 方法来自于 java. awt 包,如果需要 Applet 显示一些内容的话,必须覆盖 paint 方法进行写屏。paint 可画出 Applet 的画面,不论所显示的内容是文字、线条、背景色,还是一幅图形,都必须在 paint 方法中完成。

在一个生命周期内,paint 方法可被多次调用。例如,Applet 初始化后首次被画出;当浏览器窗口被覆盖后又重新显示出来时将自动调用 paint 方法;当浏览器窗口发生移动时也将自动调用 paint 方法;当 Applet 中包含一个动画时更是需要多次写屏。

paint 方法需要一个参数,它是一个 Graphics 对象即图形界面,可在上面以图形方式显示任何内容。Graphics 对象是由浏览器建立并自动传递给 paint 的,因此不必担心,但是一定要在程序中引用 java. awt. Graphics 类。

### 7.1.3 HTML 和 Applet 的参数传递

从 HTML 文件向 Applet 传递参数为用户提供了一个强有力的手段,通过参数传递大大加强了 Applet 的灵活性,使你的 Applet 不用修改和重新编译就能适用于多种情形。

HTML 语言有一个<APPLET>标签,可加入各种属性来指定与 Applet 有关的内容,其中 CODE、WIDTH、HEIGHT 是必选属性。如果需要向 Applet 传递参数,可通过 PARAM 标签来指定。这样做的好处是:Applet 一次编译成功就可保持不变,如果需要 Applet 显示不同的内容,仅修改 HTML 文件中指定的参数即可。

例 7.2 Applet 的参数传递,如图 7.3 所示。

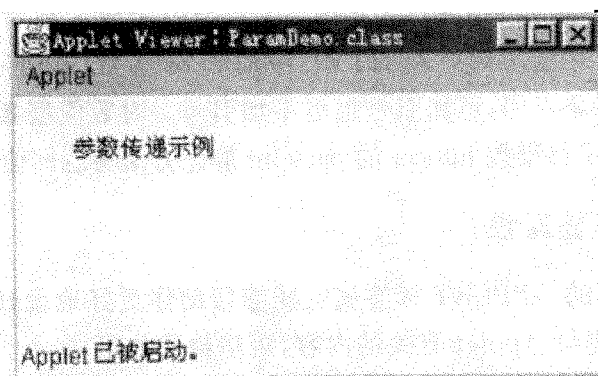


图 7.3

```
import java.applet. Applet;
import java.awt. Graphics;

public class ParamDemo extends Applet {
    String string=null;
    int x, y;

    public void init() {
        string=getParameter("message");
        x= Integer.parseInt(getParameter("xPos"));
        y= Integer.parseInt(getParameter("yPos"));
    }

    public void paint(Graphics g) {
        if (string! =null)
            g.drawString(string, x, y);
    }
}
```

相应的 HTML 文件内容如下:

```
<APPLET
  codebase=d:/mydoc/ch7
  code=ParamDemo.class
  width=300
  height=120>
  <PARAM name=message value="参数传递示例">
  <PARAM name=xPos value=30>
  <PARAM name=yPos value=30>
</APPLET>
```

本例定义了 3 个变量用来接受 HTML 传递的参数,通过调用 Applet 类的

getParameter方法接受参数。其中 string 接受一个字符串、x 和 y 分别接受显示位置。getParameter 的参数必须和 HTML 中由 name 指定的参数名相对应,然后返回由 value 指定的参数值。如程序中的 x 对应着 HTML 中的 xPos 参数,返回值为 30。注意:HTML 中的参数值都被定义为字符串,可加引号也可不加引号。对数值型参数需要调用相应的方法进行转换,程序调用了整型数 Integer 的 parseInt 静态方法将返回值转换为整型数。

#### 7.1.4 APPLET 标签属性

熟悉 HTML 语言的 APPLET 标签属性能够帮助你更好地设计出 Applet,该标签可用大写或小写符号指定与 Applet 有关的内容,语法格式如下:

```
<APPLET
  CODE=字节码文件名
  WIDTH=宽度
  HEIGHT=高度
  [CODEBASE=字节码文件路径]
  [ALT=可替换的文本内容]
  [NAME=对象名]
  [ALIGN=对齐方式]
  [VSPACE=垂直间隔]
  [HSPACE=水平间隔]>
  [<PARAM NAME=参数名 VALUE=参数值>]
  ...
  [alternateHTML]
</APPLET>
```

注意:CODE、WIDTH、HEIGHT 是必选属性,属性可用大写或小写符号表示,属性值定义为字符串,但可省略引号,所有属性必须包含在<APPLET>和</APPLET>之间。下面我们讨论这些属性的用法。

(1) CODE 用来指定 Applet 字节码文件名,可省略扩展名。

(2) WIDTH 和 HEIGHT 用来指定 Applet 显示区域的大小,以像素点为计量单位。

(3) CODEBASE 用来指定 Applet 字节码文件路径。当字节码文件和 HTML 文件不在同一个目录下时,应指定字节码文件的位置,可采用 URL 格式即网络地址。对于本地计算机可采用绝对路径,但必须注意格式,请参考例 7.2。

(4) ALT 用来指定替换显示的文本内容。如果浏览器不能运行 Applet,就显示替换文本内容,例如:ALT="指定的字节码文件找不到"。

(5) NAME 用来指定 Applet 的实例化对象名,使同一个 Web 页上的多个 Applet 可以互相识别出来。

(6) ALIGN 用来指定 Applet 在浏览器窗口中的对齐方式。常用值有 left(左对齐),right(右对齐),top(靠上),middle(对中),bottom(靠下)等。

(7) VSPACE 和 HSPACE 用来指定 Applet 四周的间隔,以像素点为计量单位。VSPACE 指定上下间隔,HSPACE 指定左右间隔。

(8) PARAM 标签包含两个参数:NAME 指定参数名,VALUE 指定参数值。Applet 可通过 getParameter 方法读取这两个参数。

(9) alternateHTML 用来指定可替换的 HTML 代码。如果你的浏览器不支持 APPLET 标签,将忽略<APPLET>和<PARAM>内容,显示指定的 HTML 代码。

### 7.1.5 Applet 与 Application 的合并运行

我们知道 Java Applet 和 Application 的区别在于运行方式的不同。那么能不能将它们合并起来,让同一个程序既可以由浏览器运行又可以单独运行呢?答案是肯定的。

**例 7.3** Applet 与 Application 合并运行。图 7.4 所示为 Applet 运行方式,图 7.5 所示为 Application 运行方式。

```
import java.applet. * ;
import java.awt. * ;
import java.awt.event. * ;

public class AppDemo extends Applet implements ActionListener {
    Button button;
    TextField field;

    public static void main(String[] args) {
        Frame window=new Frame("AppDemo"); // 创建窗口对象
        AppDemo app=new AppDemo(); // 创建程序对象
        window.add("Center", app); // 将程序对象添加到窗口
        app.init(); // 调用程序的初始化方法
        window.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        }); // 以上用匿名类的方式为窗口添加关闭功能,参见第 9 章
        window.setSize(300,120); // 设定窗口大小
        window.setVisible(true); // 设定窗口可见
    }

    public void init() {
        button=new Button("显示");
        button.addActionListener(this);
        field=new TextField(23);
        add(field);
        add(button);
    }

    public void actionPerformed(ActionEvent e) {
        field.setText("Applet 与 Application 的合并运行");
    }
}
```

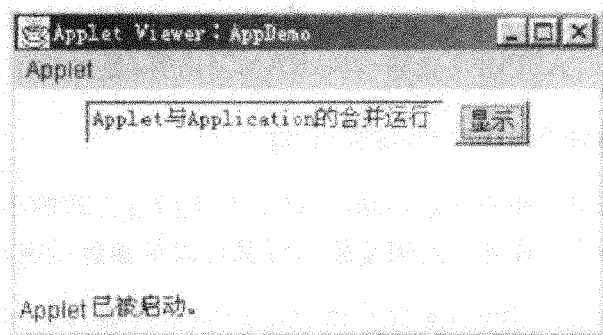


图 7.4

从程序结构上看, Applet 必须从 `java.applet.Applet` 继承, 而 `Application` 则必须有一个公共方法 `main`。另一方面, 两者的主线程也不同, Applet 由 `init` 方法进行初始化工作, 而 `Application` 则由 `main` 方法启动程序。由于上述差别的存在, 编写 Applet 和 `Application` 合并运行的程序必须遵守一定的规则。

首先, 程序应该是 `java.applet.Applet` 的子类, 这是以 Applet 方式运行的必要条件。如果程序设计成 `Frame` 的子类就无法以 Applet 方式运行。其次, 需要生成程序的一个实例对象, 通过调用对象的 `init` 方法进行初始化。

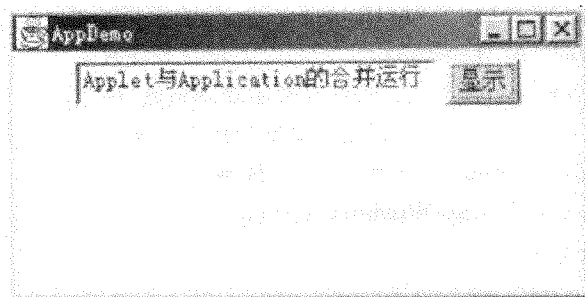


图 7.5

上例在 `main` 方法中先创建了一个 `Frame` 对象 `window`, 因为图形化的 `Application` 必须要有一个窗口。然后调用默认构造方法创建实例对象 `app`, 将 `app` 添加到 `window` 中并调用其 `init` 方法进行初始化。最后, 按照 Applet 的要求设计 `init` 方法和其他必需的方法。

以 Applet 方式运行时, 程序将忽略 `main` 方法, 因其他部分是一个完整的 Applet, 所以程序可以正常运行。以 `Application` 方式运行时, 则在 `main` 方法中创建对象并调用 `init` 方法完成初始化, 也可以正常运行。

## 7.2 字符串类

字符串是程序设计中经常用到的数据结构, 很多编程语言将字符串定义为基本数据



类型。但在 Java 语言中,字符串被定义为一个类,无论是字符串常量还是变量,都必须先生成 String 类的实例对象然后才能使用。

java.lang 有两个字符串类 String 和 StringBuffer,封装了字符串的全部操作。其中 String 用来处理创建以后不再改变的字符串,StringBuffer 用来处理可变字符串。

### 7.2.1 字符串与字符串类

字符串是一个完整的字符序列,可以包含字母、数字和其他符号。在 Java 中,用双引号括起来的字符串是字符串常量,又称为无名字符串对象,由 Java 自动创建。字符串常量可以赋给任何一个 String 对象引用,这样处理从表面上看起来和其他编程语言没有大的差别,照顾了程序员的习惯,但实际上存在着较大的差异。无论何时,Java 中的字符串都是以对象的面孔出现的,在运行时要为它分配内存空间,创建对象引用。

Java 将字符串定义为类有哪些好处呢?

首先,在任何系统平台上都能保证字符串本身以及对字符串的操作是一致的。对于网络环境,这一点是至关重要的。其次,String 和 StringBuffer 经过了精心设计,其功能是可以预见的。为此,二者都被说明为最终类,不能派生子类,以防用户修改其功能。最后,String 和 StringBuffer 类在运行时要经历严格的边界条件检验,它们可以自动捕获异常,提高了程序的健壮性。

下面这个程序使用 String 和 StringBuffer 类实现了字符串的翻转。

例 7.4 字符串的翻转,如图 7.6 所示。

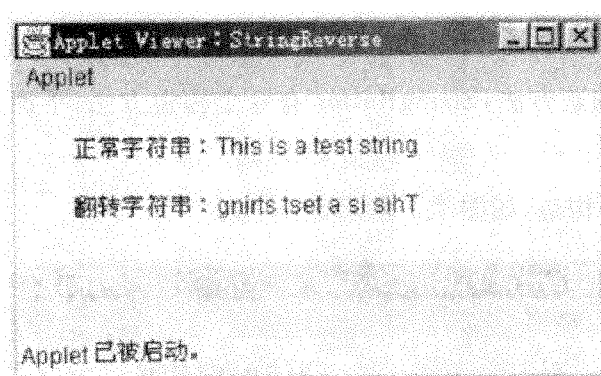


图 7.6

```
import java.applet.Applet;
import java.awt.Graphics;

public class StringReverse extends Applet {
    public void paint(Graphics g) {
        String str="This is a test string";
        g.drawString("正常字符串:"+str, 30, 30);
        g.drawString("翻转字符串:"+reverse(str), 30, 60);
    }
}
```

```

public String reverse(String s) {
    int len=s.length();
    StringBuffer buffer=new StringBuffer(len);
    for (int i=len-1; i>=0; i--)
        buffer.append(s.charAt(i));
    return buffer.toString();
}
}

```

语句 `String str="This is a test string"` 将字符串常量赋给了 `str`。实际上,Java 先为字符串常量创建了无名字符串对象,然后才把它的引用赋给 `str`。

在 `drawString` 方法中,字符串常量和字符串变量进行了“+”运算,其结果仍为字符串对象,字符串连接是 `String` 类的唯一运算。程序首先输出了原始字符串,然后输出了翻转字符串,对字符串的翻转操作在 `reverse` 方法中实现。

在 `reverse` 方法中,`buffer` 是 `StringBuffer` 对象,在创建时指定其为参数 `s` 的长度。`s` 继承了 `String` 类的方法,`s.charAt(i)` 可返回字符串的第 `i` 个字符。`StringBuffer` 有专门的字符串连接方法 `append`,作用是将参数值添加到对象尾部。在 `for` 循环中,每次反向从 `s` 中取出一个字符添加到 `buffer` 的尾部。循环完毕,`buffer` 中存放的就是翻转后的字符串。

注意:`reverse` 方法的返回值是 `String` 类型,因此调用 `toString` 方法将 `StringBuffer` 类型转换成 `String` 类型再返回。

### 7.2.2 字符串类的构造方法

`String` 类有 9 个构造方法,`StringBuffer` 有 3 个构造方法,下面通过一个例子介绍它们的用法。

**例 7.5** 字符串的构造,如图 7.7 所示。

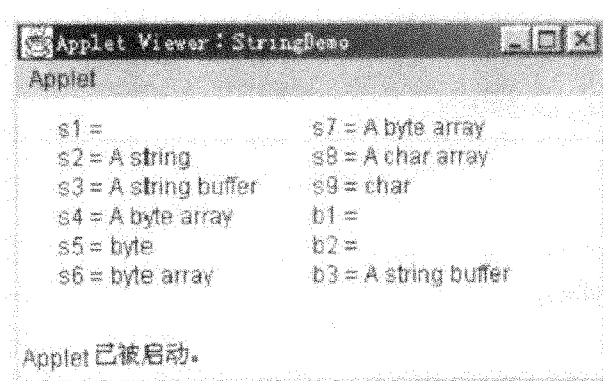


图 7.7

```

import java.io. * ;
import java.applet. Applet;
import java.awt. Graphics;

```

```

public class StringDemo extends Applet {
    byte b[]={'A', ' ', 'b', 'y', 't', 'e', ' ', 'a', 'r', 'r', 'a', 'y'};
    char c[]={'A', ' ', 'c', 'h', 'a', 'r', ' ', 'a', 'r', 'r', 'a', 'y'};
    String s1, s2, s3, s4, s5, s6, s7, s8, s9;
    StringBuffer b1, b2, b3;

    public void init() {
        b1=new StringBuffer();    // 创建一个空 StringBuffer 对象
        b2=new StringBuffer(10); // 创建长度为 10 的空 StringBuffer 对象
        b3=new StringBuffer("A string buffer"); // 以字符串为参数创建 StringBuffer 对象
        s1=new String();          // 创建一个空 String 对象
        s2=new String("A string"); // 以字符串为参数创建 String 对象
        s3=new String(b3);        // 以 StringBuffer 对象为参数创建 String 对象
        s4=new String(b);         // 以 b 为参数创建 String 对象,8 位字节自动转为 16 位字符
        s5=new String(b,2,4);     // 从 b 的第 3 位,取 4 个元素为参数创建 String 对象
        try {                     // 如果下面的字符集编码不存在将抛出异常
            s6=new String(b,2,10,"GBK"); // 同 s5,最后的字符串参数为字符集编码
            s7=new String(b,"GBK");      // 同 s4,最后的字符串参数为字符集编码
        } catch (UnsupportedEncodingException e) {} // 捕获异常
        s8=new String(c);         // 以字符数组 c 为参数创建 String 对象
        s9=new String(c,2,4);     // 从 c 的第 3 位,取 4 个元素为参数创建 String 对象
    }

    public void paint(Graphics g) {
        g.drawString("s1 = "+s1, 20, 20);
        g.drawString("s2 = "+s2, 20, 35);
        g.drawString("s3 = "+s3, 20, 50);
        g.drawString("s4 = "+s4, 20, 65);
        g.drawString("s5 = "+s5, 20, 80);
        g.drawString("s6 = "+s6, 20, 95);
        g.drawString("s7 = "+s7, 150, 20);
        g.drawString("s8 = "+s8, 150, 35);
        g.drawString("s9 = "+s9, 150, 50);
        g.drawString("b1 = "+b1.toString(), 150, 65);
        g.drawString("b2 = "+b2.toString(), 150, 80);
        g.drawString("b3 = "+b3.toString(), 150, 95);
    }
}

```

上例定义了一个字节型数组 b 和一个字符型数组 c,并分别赋值。在 init 方法中首先调用 StringBuffer 类的 3 个构造方法创建了 3 个对象,然后调用 String 类的 9 个构造方法创建了 9 个对象。paint 方法显示了这些字符串对象的内容。

注意:构造方法的含义参见程序注释,其中用到的“GBK”是中文版 Windows 的默认字符集编码,另一个常用的字符集编码是“ASCII”。

### 7.2.3 String 类的应用

String 类提供了很多方法,可对字符串进行各种处理。限于篇幅,下面只介绍其中的常用方法。

#### 1. 求字符串长度

public int length() 可返回字符串长度。例如:

```
String s="欢迎使用 Java 语言";  
int len=s.length();
```

len 的值为 10。注意:Java 采用 Unicode 编码,每个字符为 16 位长,因此汉字和其他符号一样只占用一个字符。另外,字符串的 length 方法和表示一个数组长度的 length 是不一样的,后者是一个实例变量。

#### 2. 字符串连接

public String concat(String str) 可返回一个字符串,它将把参数 str 添加在原字符串的后边。例如:"to".concat("get").concat("her") 的返回值为"together"。

但在 Java 中,更多的是用"+"来连接字符串。例如:

```
String str="hello";  
str=str+" world!";
```

这里你可能会产生一个小小的迷惑,因为我们前边讲过 String 对象是不能改变的。但实际上字符串连接是由编译器利用 StringBuffer 来完成的,上例相当于:

```
String str="hello";  
str=new StringBuffer().append(str).append(" world!").toString();
```

另外,"+"还可以把一个非字符串的数值连接到字符串尾部,编译器将自动把非字符串转换为字符串再进行连接。

#### 3. 字符串截取

字符串截取有两个途径:一次截取一个字符或一次截取一个子串。前者可通过 charAt 方法(参见例 7.4),后者使用 substring 方法。它有两种形式:

```
String substring(int start)  
String substring(int start, int end)
```

其中 start 代表起始位置,end 代表结束位置。例如:

```
String str="a short string";  
String s1=str.substring(2);  
String s2=substring(8, 14);
```

则 s1 的内容为“short string”,s2 的内容为“string”。注意:Java 中字符串的起始位置从 0 开始,即第一个字符的位置是 0,最后一个字符的位置是字符串长度减 1。substring 的第二个参数实际上是起始位置加上子串长度。

#### 4. 字符串比较

有两组方法用于字符串比较,一组是 equals,用于比较两个字符串是否相等,返回值为布尔值。一组是 compareTo,用于按字符顺序比较两个字符串,返回值为整型数。共有 5 个方法:

```
boolean equals(Object object)
boolean equalsIgnoreCase(String str) // 忽略字符大小写
int compareTo(Object object)
int compareTo(String str)
int compareToIgnoreCase(String str) // 忽略字符大小写
```

例如:

```
String s1="This";
String s2="That";
System.out.println("结果 1:"+s1.equals("this")+" "+s1.equalsIgnoreCase("this"));
System.out.println("结果 2:"+s2.compareTo("that")+" "+s2.compareToIgnoreCase("that"));
```

屏幕输出为:

```
结果 1:false, true
结果 2:-32, 0
```

其中,s2.compareTo("that") 的返回值为-32,说明“That”小于“that”(按字母序 T 小于 t)。如果一个字符串某位置上的字符大于另一个字符串对应位置上的字符,则比较结果为大于 0 的整数。只有当两个字符串完全相等即长度一致、位置一致、大小写一致,比较结果才为 0。

#### 5. 拷贝到字符串

一个字符数组的内容可以全部或部分地拷贝到一个字符串中。有两个静态方法用于这种拷贝:

```
static String copyValueOf(char[] data)
static String copyValueOf(char[] data, int offset, int count)
```

例如:

```
char c[]={'A',' ','c','h','a','r',' ','a','r','r','a','y'};
String s1=new String(), s2=new String();
s1=s1.copyValueOf(c);
s2=s2.copyValueOf(c, 2, 4);
```

则 s1 的值为“A char array”,s2 的值为“char”。注意第二种形式,2 是指字符数组中被拷贝部分的起始位置,4 是指拷贝的字符个数。

## 6. 字符串大小写转换

一个字符串可以整体转换为大写或小写字符,例如:

```
String s1="all is lowercase";  
String s2="Some Is Uppercase";  
s1=s1.toUpperCase();  
s2=s2.toLowerCase();
```

结果是 s1 的字符全部为大写,s2 的字符全部为小写。

## 7. 字符串检索

你可以在一个字符串中检索指定字符或子串的位置,如果检索到将返回一个代表位置的整数,否则返回值为-1。有两组方法实现这种操作,indexOf 方法返回字符或子串首次出现的位置,lastIndexOf 方法返回字符或子串最后一次出现的位置。我们各选择两个典型的方法来说明这种检索操作:

```
int indexOf(int ch)  
int indexOf(String str)  
int lastIndexOf(int ch)  
int lastIndexOf(String str)
```

例如:

```
String str="This is a test string";  
int i=str.indexOf('i');  
int j=str.lastIndexOf('i');  
int k=str.indexOf("is");  
int l=str.lastIndexOf("is");
```

最后的结果是:i 为 2、j 为 18、k 为 2、l 为 5。

## 8. 字符串转换为数组

字符串可以转换为字节数组或字符数组,这种转换在第 12 章介绍的 Java 流处理中十分有用。字符串转为字节数组将进行特别处理,因为字符是 16 位长,而字节为 8 位长,所以要将字符的高 8 位去掉,只保留低 8 位成为一个字节。有 3 个方法:

```
byte[] getBytes()           // 按系统默认字符集编码转换为字节数组  
byte[] getBytes(String enc) // 其中 enc 为字符集编码,参见例 7.5  
char[] toCharArray()        // 转换为字符数组
```

例如:

```
byte byteArr[];
```

```

char charArr[];
String str="This is a test string";
byteArr=str.getBytes();
charArr=str.toCharArray();

```

注意:可以只声明数组而不创建,在转换过程中由系统自动创建。也可以直接创建数组,数组长度可以是 0 或大于 0 的整数。

## 9. 转换为字符串

String 类提供了一组 valueOf 方法用来将其他数据类型转换成字符串,其参数可以是任何数据类型(byte 类型除外)。它们都是静态的,也就是说不必创建实例化对象即可直接调用这些方法,其基本用法为:valueOf(数据类型)。例如:

```

char data[]={'a','b','c','d','e'};
System. Out. println(String. valueOf(12D));
System. Out. println(String. valueOf(3<2));
System. Out. println(String. valueOf(data,1,3));

```

输出结果为:

```

12.0
false
bcd

```

### 7.2.4 StringBuffer 类的应用

StringBuffer 提供的方法有一些与 String 相同,有一些不同。最主要的方法有两组,一组是 append,另一组是 insert,每组各有 10 个方法。

#### 1. append 方法

append 的 10 个方法主要在参数上有所不同,它可以把各种数据类型转换成字符串后添加进来(byte 类型除外),其基本用法为:append(数据类型)。例如:

```

char data[]={'a','b','c','d','e'};
StringBuffer buffer=new StringBuffer();
buffer.append(100);
buffer.append(' ');
buffer.append(2.5F);
buffer.append(" is equal to ");
buffer.append(250.0D);
buffer.append(' ');
buffer.append(data);
buffer.append(' ');
buffer.append(data, 2, 3);

```

最后,buffer 的内容为“100 \* 2.5 is equal to 250.0 abcde cde”。注意:输出 buffer 时可调用 toString 方法将其转换为字符串。

## 2. insert 方法

insert 方法和 append 方法在使用上非常类似,唯一的不同是多了一个位置参数,该参数必须大于等于 0。其基本用法为:insert(插入位置,数据类型)。例如:

```
char data[]={'a','b','c','d','e'};
StringBuffer buffer=new StringBuffer();
buffer.insert(0, 100);
buffer.insert(0, 2.5F);
buffer.insert(3, '*');
buffer.insert(0, 250.0D);
buffer.insert(5, " is equal to ");
```

最后,buffer 的内容为“250.0 is equal to 2.5 \* 100”。

## 3. 其他方法

下面几个方法对程序员来说是经常用到的:

```
public StringBuffer delete(int start, int end) // 删除子串
public StringBuffer deleteCharAt(int index) // 删除指定位置上的字符
public StringBuffer replace(int start, int end, String str) // 替换子串
StringBuffer reverse() // 翻转字符串
```

例如:

```
StringBuffer buffer=new StringBuffer("This is a test string");
buffer=buffer.delete(0, 8);
buffer=buffer.deleteCharAt(0);
buffer=buffer.replace(0, 1, "This is a");
buffer=buffer.reverse();
```

结果为:

```
a test string
test string
This is a test string
gnirts tset a si si hT
```

## 7.3 标准输入/输出

Java 中的系统类 System 是一个最终类,所有成员变量和方法都是静态的,因此你可以在程序中直接调用它们。System 的一个重要功能就是标准输入/输出。一般情况下,



数据输入的来源为键盘,输出的目的地是屏幕,通过调用各种标准输入/输出方法来实现。

### 7.3.1 标准输入方法

键盘是标准输入设备,用户通常用键盘来输入数据。System 类含有标准输入流的成员变量 in,我们可以调用它的 read 方法来读取键盘数据。read 方法有 3 种格式:

```
public abstract int read()
public int read(byte[] b)
public int read(byte[] b, int off, int len)
```

**例 7.6** 一次从键盘读入一个字符,如图 7.8 所示。

```
class ReadOne {
    public static void main(String[] args) throws java.io.IOException {
        char ch;
        StringBuffer str=new StringBuffer();
        System.out.println("\n从键盘输入字符,按回车键结束。");
        while((ch=(char)System.in.read())!='\r')
            str.append(ch);
        System.out.println(str.toString());
    }
}
```

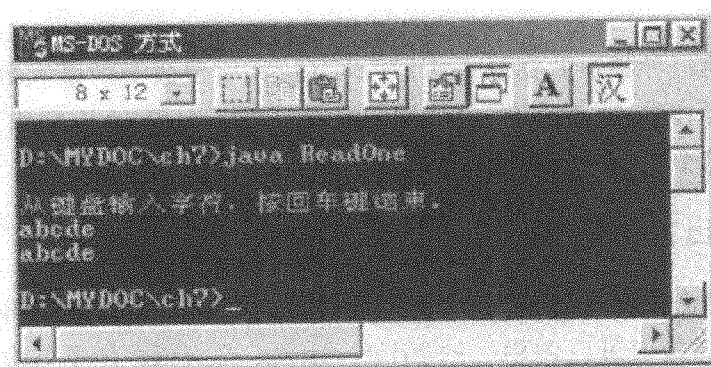


图 7.8 一次从键盘读入一个字符

上例使用了第一种读取数据的方法。程序创建了一个 StringBuffer 对象 str,在循环中重复调用 read 方法读取字符,并将其添加到 str。当用户按下回车键时,相当于输入了一个字符“\r”,此时就结束循环。最后将输入的字符显示出来。

需要注意的是,read 的返回值为整型数,因此在将数据赋给字符变量 ch 前,要转换成字符类型。另外,read 会产生输入异常,要么放在 try...catch 块中执行,要么令 main 方法将异常上交(即在声明语句中加入 throws IOException),这样才能通过编译。

**例 7.7** 一次从键盘读入多个字符。

```
class ReadMore {
    public static void main(String[] args) throws java.io.IOException {
```

```

byte data[] = new byte[40];
System.out.println("\n从键盘输入不超过40个字符,按回车键结束。");
System.in.read(data);
System.out.write(data, 0, data.length);
}
}

```

上例使用了第二种读取数据的方法。程序创建了一个字节数组 data 用来保存键盘输入的数据,最多只能接收 40 个字符,用户一旦按下回车键就结束输入。

需要注意的是,读取的数据保存在字节数组中,如要输出字节数组的内容,可先将其转换为字符串然后再显示出来。本例调用了 write 方法在标准输出设备上显示字节数组的内容,如果调用 println 方法则不能正常显示。

**例 7.8** 一次从键盘读入不超过固定数值的字符。

```

class ReadSome {
    public static void main(String args[]) throws java.io.IOException {
        byte data[] = new byte[40];
        System.out.println("\n从键盘输入字符,按回车键结束。");
        System.in.read(data, 0, 10);
        System.out.write(data, 0, data.length);
    }
}

```

上例使用了第三种读取数据的方法,对 read 读入字节数组的最大数加以限制。程序创建的字节数组可容纳 40 个字节,但读取的数据被限定在前 10 个数组元素中。所以,当键入的数据多于这个数值时,多余的数据就会被 read 方法丢弃。

### 7.3.2 标准输出方法

屏幕是标准输出设备,用户可以将数据在屏幕上显示出来。System 类含有标准打印流的成员变量 out,我们可以调用它的 print、println 或 write 方法来输出各种类型的数据。标准输出方法和标准输入方法不同,它们不产生输出异常。另外,在输出的过程中,所有数据都按照系统字符集编码转换成字节。

print 和 println 的参数完全一样,不同之处在于 println 输出后换行而 print 不换行。下面,我们只给出 println 和 write 的格式:

```

void println()           // 输出一个换行符
void println(boolean x)  // 输出一个布尔值 true 或 false
void println(char x)     // 输出一个字符
void println(char[] x)   // 输出一个字符数组
void println(double x)   // 输出一个双精度值
void println(float x)    // 输出一个浮点值
void println(int x)       // 输出一个整型值
void println(long x)      // 输出一个长整型值

```

```

void println(Object x)          // 输出一个对象的字符表示
void println(String x)         // 输出一个字符串
void write(int b)               // 输出一个字节
void write(byte[] b, int offset, int length) // 输出字节数组的一部分

```

**例 7.9** 标准输出方法的使用,如图 7.9 所示。

```

class PrintDemo {
    public static void main(String args[]) {
        Object o="an object";
        char c[]={'a', 'b', 'c', 'd', 'e'};
        byte b[]={'f', 'g', 'h', 'i', 'j'};

        System.out.println(true);
        System.out.println('C');
        System.out.println(1000);
        System.out.println(1000000L);
        System.out.println(12.5F);
        System.out.println(12345.99D);
        System.out.println("a string");
        System.out.println(o);
        System.out.println(c);
        System.out.write(b,0,b.length);
        System.out.println();
        System.out.write(b[0]);
        System.out.flush();
    }
}

```

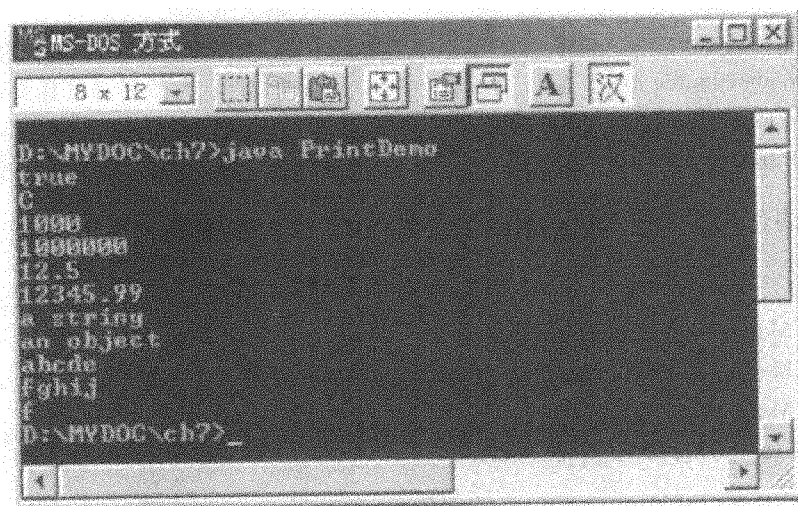


图 7.9

注意程序中对 Object 的定义,此时它相当于一个字符串。write 在输出时不换行,因

此程序调用 `println()` 方法输出换行符。另外, `write` 方法在输出单个字节时, 并不能立即显示出来, 必须调用 `flush` 方法或 `close` 方法强制回显。

## 7.4 其他常用类

本节介绍 Java 语言的 6 个常用类, 它们是数学函数类 `Math`, 提供了基本数学运算; 日期类 `Date`、`Calendar` 和 `DateFormat`, 提供了日期和时间操作; 随机数类 `Random`, 提供了随机数生成器; 向量类 `Vector`, 提供了类似于可变长数组的操作。

### 7.4.1 数学函数类 `Math`

`Math` 是一个最终类, 含有基本数学运算函数, 如指数运算、对数运算、求平方根和三角函数等, 可以直接在程序中加 `Math` 前缀调用。下面是其成员变量和常用成员方法:

<code>static double E</code>	// 数学常量 e
<code>static double PI</code>	// 圆周率 $\pi$
<code>static double sin(double a)</code>	// 正弦函数, 参数为弧度
<code>static double cos(double a)</code>	// 余弦函数, 参数为弧度
<code>static double tan(double a)</code>	// 正切函数, 参数为弧度
<code>static double toDegrees(double angdeg)</code>	// 弧度转换为角度
<code>static double toRadians(double angdeg)</code>	// 角度转换为弧度
<code>static double exp(double a)</code>	// 常数 e 的 a 次幂
<code>static double log(double a)</code>	// 自然对数
<code>static double sqrt(double a)</code>	// 平方根
<code>static double pow(double a, double b)</code>	// 数 a 的 b 次方
<code>static int round(float a)</code>	// 四舍五入
<code>static long round(double a)</code>	// 四舍五入
<code>static double random()</code>	// 大于等于 0.0 小于 1.0 的随机数
<code>static double abs(double a)</code>	// 绝对值, 参数还可以是 float、int、long
<code>static double max(double a, double b)</code>	// 最大值, 参数还可以是 float、int、long
<code>static double min(double a, double b)</code>	// 最小值, 参数还可以是 float、int、long

这些方法的使用都比较简单, 只要传递正确的参数就可得到正确的返回值。例如下面这段程序:

```
double d1=Math.sin(Math.toRadians(30.0));
double d2=Math.log(Math.E);
double d3=Math.pow(2.0, 3.0);
int r=Math.round(33.6F);
```

结果是: d1 为 0.49999999999999994、d2 为 1.0、d3 为 8.0、r 为 34。

### 7.4.2 日期类

Java 提供了 3 个日期类: `Date`、`Calendar` 和 `DateFormat`, 早期版本定义的 `Date` 和

Time 类中的大部分方法在 JDK1.3 中已不再使用。在程序中,对日期的处理主要是如何获取、设置和格式化,Java 的日期类提供了很多方法以满足程序员的各种需要。其中,Date 主要用于创建日期对象并获取日期,Calendar 可获取和设置日期,DateFormat 主要用来创建日期格式化器,由格式化器将日期转换为各种日期格式串输出。

Java 语言规定的基准日期为 1970.1.1.00:00:00 格林威治标准时,当前日期是由基准日期开始所经历的毫秒数转换出来的。由于 Java 是网络语言,程序必须注意到各个国家对日期格式的不同理解。

**例 7.10** 日期的获取、设置和格式化,如图 7.10 所示。

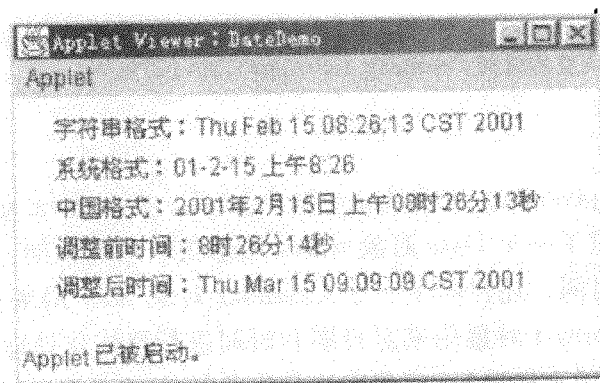


图 7.10

```
import java.text. * ;
import java.util. * ;
import java.awt. * ;
import java.applet. * ;

public class DateDemo extends Applet {
    public void paint(Graphics g) {
        Date today;
        Calendar now;
        DateFormat f1,f2;
        String s1,s2;

        today=new Date(); // 获取系统当前日期
        g.drawString("字符串格式:" + today.toString(),20,20);

        f1=DateFormat.getInstance(); // 以默认格式生成格式化器
        s1=f1.format(today); // 将日期转换为字符串
        g.drawString("系统格式:" + s1,20,40);

        // 生成长格式的中国日期格式化器
        f1=DateFormat.getDateInstance(DateFormat.LONG, Locale.CHINA);
        // 生成长格式的中国时间格式化器
```

```

f2=DateFormat.getInstance(DateFormat.LONG, Locale.CHINA);
s1=f1.format(today); // 将日期转换为日期字符串
s2=f2.format(today); // 将日期转换为时间字符串
g.drawString("中国格式:" + s1 + " " + s2, 20, 60);

now=Calendar.getInstance(); // 获取系统时间
s1=now.get(now.HOUR)+"时"
    +now.get(now.MINUTE)+"分"
    +now.get(now.SECOND)+"秒";
g.drawString("调整前时间:" + s1, 20, 80);
now.set(2001, 2, 15, 9, 9, 9);
today=now.getTime();
g.drawString("调整后时间:" + today.toString(), 20, 100);
}
}

```

DateFormat 类在 java.text 包中, Date 和 Calendar 类在 java.util 包中, 程序必须引入这两个包。上例创建了一个 Date 对象 today 用来获取系统日期, 实例化以后, today 就被放入了系统日期和时间。还声明了一个 Calendar 对象 now 用来获取和设置日期和时间, 声明了两个 DateFormat 对象用来对日期和时间进行格式化处理。

today 中的日期不能直接输出, 一种方法是将其转换成字符串输出, 从图中可以看到字符串日期格式包含了较多的内容。如果字符串格式不能满足程序的需要, 可采用第二种方法即通过日期格式化器进行转换。程序先调用 DateFormat 的 getInstance 方法创建了一个默认格式化器, 它可以同时对日期和时间进行格式化, 转换工作由 DateFormat 的 format 方法完成, 它的参数为一个日期对象, 返回值为字符串类型。

DateFormat 还提供了对日期和时间分开进行处理的格式化器, 它们可以接收更多的参数, 对日期或时间进行多种格式化处理。其中一个参数决定了格式串的长度, 已定义的静态变量有 SHORT、MEDIUM、LONG、FULL、DEFAULT。另外一个参数决定了日期或时间的国家格式, Locale 类中有世界主要国家的变量定义。例如, 中国定义为 CHINA, 美国是 US、英国是 UK、德国为 GERMAN 等等。用给定参数生成格式化器, 再调用该格式化器的 format 方法, 就可将日期转换成相应的字符串输出。

最后, 程序调用了 Calendar 的 getInstance 方法对 now 进行实例化, 完成以后, 系统日期和时间就被保存到该对象的各个成员变量中。如果你需要对日期和时间的各个分量进行处理, Calendar 类就是最佳选择, 它提供了 get 和 set 方法获取和设置日期分量, 还提供了 add 和 roll 方法对日期分量进行运算。程序调用了 get 方法获取时间分量, 并将它们合成为字符串输出。然后调用 set 方法重新设定年、月、日、时、分、秒, 调用 getTime 方法整体取出日期数据保存到 today 中, 转换成字符串后输出。从图中可以观察到改变后的日期和时间。注意: 此时仅仅是改变了 now 的日期分量值, 系统时钟并没有被改变, 因为 Java 不允许 Applet 这样做。

### 7.4.3 随机数类 Random

Math 中的 random 方法可产生一个伪随机数, 这种方式比较简单。为了适应网络时

代编程对随机数的需要,Java 在 Random 类中提供了更多的功能,Random 的实例化对象可使用一个 48 位长的种子数来生成随机数。如果两个 Random 对象使用同样的种子数,并以同样的顺序调用生成方法,仍然可以保证得到两个不同的 32 位伪随机数。为了使 Java 程序有良好的可移植性,应该尽可能使用 Random 类来生成随机数。

Random 有两个构造方法:Random()、Random(long seed)。前者使用系统时间作为种子数,后者使用指定的种子数。构造方法只是创建了随机数生成器,必须调用生成器的方法才能产生随机数。

**例 7.11** 生成各种类型的随机数,如图 7.11 所示。

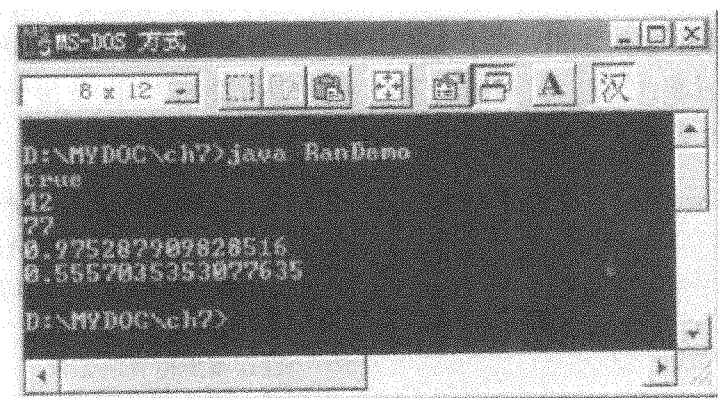


图 7.11

```
import java.util.*;  
class Randemo {  
    public static void main(String args[]) {  
        Random r1=new Random(1234567890L);  
        Random r2=new Random(1234567890L);  
  
        boolean b=r1.nextBoolean(); // 随机数不为 0 时取真值  
        int i1=r1.nextInt(100);      // 产生大于等于 0 小于 100 的随机数  
        int i2=r2.nextInt(100);      // 同上  
        double d1=r1.nextDouble();  // 产生大于等于 0.0 小于 1.0 的随机数  
        double d2=r2.nextDouble();  // 同上  
        System.out.println(b);  
        System.out.println(i1);  
        System.out.println(i2);  
        System.out.println(d1);  
        System.out.println(d2);  
    }  
}
```

上例创建了两个随机数生成器,使用的种子数相同,从图 7.10 可以看出由这两个生成器产生的随机数是不同的。程序中生成了 boolean、int 和 double 型随机数,Random 类还可以生成其他类型的随机数,如 long、float 等。注意:Random 包含在 java.util 包中,

程序需要引入该包。

#### 7.4.4 向量类 Vector

大多数编程语言中的数组是固定长度的,即数组一经建立就不能在使用过程中改变其长度。有些程序可能要解决数组长度不断改变的问题,这对于无法预见使用多长的数组更合适的程序员来说是一件头痛的事情,Java 引入的 Vector 类较好地解决了这个问题。

Vector 被设计成一个能不断增长的序列,用来保存对象引用。在创建 Vector 对象时可以指定初始容量和增量,每次添加元素都将使向量长度按增量自动增长。

Vector 类似于可变长数组,但功能更加强大,任何类型的对象都可以放入 Vector。通过调用 Vector 封装的方法,可以随时添加或删除向量元素,以及增加或缩短向量序列的长度。Vector 的常用方法如下:

Vector(int initCapacity, int increment)	// 构造方法,指定初始容量和增量
void addElement(Object obj)	// 在尾部添加元素,长度自动增 1
boolean removeElement(Object obj)	// 删除第一个与 obj 相同的元素
void setElementAt(Object obj, int index)	// 替换指定位置上的元素
Object elementAt(int index)	// 返回指定位置的元素
int indexOf(Object obj)	// 返回指定元素 obj 在向量中的位置
int size()	// 返回元素个数
int capacity()	// 返回向量长度
void trimToSize()	// 按当前元素个数缩减向量长度
Enumeration elements()	// 生成一个向量元素的枚举

Vector 包含在 java.util 包中,下面的例子演示了 Vector 的应用。

**例 7.12** 演示 Vector 的各种用法,如图 7.12 所示。

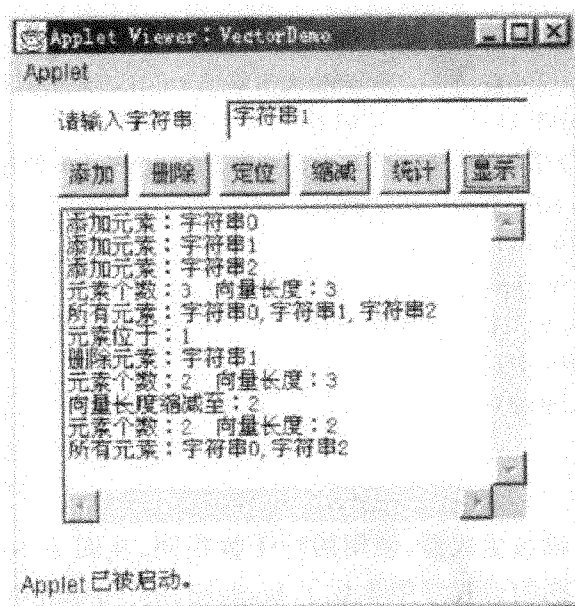


图 7.12



```

import java.util. * ;
import java.applet. * ;
import java.awt. * ;
import java.awt.event. * ;

public class VectorDemo extends Applet implements ActionListener {
    Vector vect;
    Label label;
    TextField field;
    TextArea area;
    Button btn1, btn2, btn3, btn4, btn5, btn6;

    public void init() {
        vect=new Vector(1,1);
        label=new Label("请输入字符串");
        field=new TextField(20);
        area=new TextArea(10,33);
        btn1=new Button("添加");
        btn2=new Button("删除");
        btn3=new Button("定位");
        btn4=new Button("增减");
        btn5=new Button("统计");
        btn6=new Button("显示");
        add(label);
        add(field);
        add(btn1);
        add(btn2);
        add(btn3);
        add(btn4);
        add(btn5);
        add(btn6);
        add(area);
        btn1.addActionListener(this);
        btn2.addActionListener(this);
        btn3.addActionListener(this);
        btn4.addActionListener(this);
        btn5.addActionListener(this);
        btn6.addActionListener(this);
    }

    public void actionPerformed (ActionEvent e) {
        if (e.getSource()== btn1) {
            vect.addElement(field.getText());
            area.append("添加元素;" + field.getText() + "\n");

```

```

    }
    else if (e.getSource() == btn2) {
        if (vect.removeElement(field.getText()))
            area.append("删除元素:" + field.getText() + "\n");
        else
            area.append(field.getText() + " 不在向量中\n");
    }
    else if (e.getSource() == btn3) {
        area.append("元素位于:" + vect.indexOf(field.getText()) + "\n");
    }
    else if (e.getSource() == btn4) {
        vect.trimToSize();
        area.append("向量长度缩减至:" + vect.size() + "\n");
    }
    else if (e.getSource() == btn5) {
        area.append("元素个数:" + vect.size() + " 向量长度:" + vect.capacity() + "\n");
    }
    else if (e.getSource() == btn6) {
        Enumeration enum = vect.elements();
        StringBuffer buffer = new StringBuffer();
        while(enum.hasMoreElements())
            buffer.append(enum.nextElement()).append(",");
        buffer.deleteCharAt(buffer.length() - 1);
        area.append("所有元素:" + buffer.toString() + "\n");
    }
}
}

```

上例创建了一个 Vector 对象 vect, 初始容量和增量都为 1。这样, 一旦向量用尽就自动追加 1 个元素。通过 6 个按钮演示了 Vector 的用法, 每个按钮对应一个功能。在程序运行过程中, 我们首先输入了 3 个字符串, 点击“添加”按钮将它们逐个添加到 vect 中。

actionPerformed 方法对所有按钮事件进行了处理, 根据事件源执行相应的代码, 并将执行结果添加到文本域显示出来。从图 7.11 中可以看到, 每添加一个元素, 向量就会增加一个长度。删除“字符串 1”后, 元素个数为 2, 向量长度为 3, 这表明最后一个元素是空值。点击“缩减”按钮后, 向量长度就缩减至非空元素个数。

程序使用枚举类 Enumeration 来显示向量的所有元素。枚举是一个对象序列, 可用它的 nextElement 方法逐个取出元素, hasMoreElements 方法可判断出序列中是否还有剩余的元素。在处理某些未知长度的序列时, 枚举是一个很好的工具。

## 习 题

7-1 试述 Java Applet 的工作原理, 其生命周期是如何划分的?

- 7-2 Java Applet 的 paint 方法是由谁调用的? 如何调用?
- 7-3 如何从 HTML 文件中向 Applet 传递参数? 这些参数是如何被接收和处理的?
- 7-4 试编写一个 Applet 程序,接收一个字符串和一个数值参数。在程序中根据数值参数设定文本框的长度,并将字符串参数显示在文本框中。
- 7-5 Java 为什么把字符串定义为类? 为什么定义了两个字符串类?
- 7-6 如何将字符串转换成数组? 如何将其他数据类型转换成字符串?
- 7-7 试编写一个 Applet 程序,接受用户输入的字符串和子串,统计子串在字符串中重复出现的次数。
- 7-8 编程实现一个无序字符串的有序输出。
- 7-9 试编写一个 Applet 程序,接受用户输入的字符串和字符,将字符串中的对应字符全部删除掉,并保持字符串的连续性和实际长度。
- 7-10 标准输入方法 read 在使用中需要注意哪些问题? 它输入的数据是何类型?
- 7-11 标准输出方法 println 能输出字节数据吗? 有哪些替代方法?
- 7-12 编程实现从标准输入设备上输入一个浮点数(输入后进行转换)。
- 7-13 编程实现利用 println 方法输出一个字节数组(输出前进行转换)。
- 7-14 试编写一个 Applet,将“\*”号按正弦曲线的一个周期显示出来。
- 7-15 试编写一个 Applet,分别显示美国和德国的短格式日期。
- 7-16 试编写一个 Applet,生成 100 个随机数,统计小于和不小于 0.5 的数各有多少。
- 7-17 试编写一个 Applet,生成 100 个 1~10 之间的随机数,统计小于和不小于 5 的数各有多少。
- 7-18 试编写一个 Applet,添加一个按钮和一个文本域。将鼠标的点击位置(x, y)一起保存到一个向量元素中,点击按钮将向量元素全部显示到文本域中。提示:鼠标事件监听器 MouseListener 中有一个方法 mousePressed,传递给它的事件 e 有两个方法 getX 和 getY 可获取点击位置,可将 x 和 y 合成为一个 Point 对象存入向量序列。



# 第8章

## 图形用户界面

GUI(graphic user interface)的中文意思是图形用户界面,如今这个词频繁地出现在各种计算机语言教科书中。Windows 操作系统就是典型的图形用户界面。在 GUI 中,用户可以看到什么就操作什么,取代了以往字符方式下知道什么才能操作什么的方式,极大的方便了用户对计算机的操作,成为当前的编程标准。

Java 语言可以编写出良好的图形用户界面,因为它提供了图形用户界面所需要的基本组件,如窗口、按钮、文本框、选择框、滚动条等,Java 类库 java.awt 包含了所有这些基本组件。本章介绍了图形用户界面基本组件的使用方法,如何使用布局管理器对组件进行管理,以及 Java 的事件处理机制。

### 8.1 组件

组件(Component)是构成 GUI 的基本要素,通过对不同事件的响应来完成和用户的交互或组件之间的交互。组件一般作为一个对象放置在容器(Container)内,一个容器就是能容纳和排列组件的对象,如 Applet、Panel、Frame 等。组件通过容器的 add 方法把自己加入到容器中。

#### 8.1.1 标签

##### 1. 创建标签

标签(Label)的功能是显示单行的字符串,可在屏幕上显示一些提示性、说明性的文字。例如,在列表框的旁边加上一个标签,说明列表框的功能。

**例 8.1** 创建标签的例子,如图 8.1。

```
import java.awt.*;  
import java.applet.Applet;  
public class LabelDemo extends Applet {  
    public void init() {  
        Label label1 = new Label();
```

```

Label label2=new Label("欢迎你使用标签!");
Label label3=new Label("这是一个写标签的例子", Label.RIGHT);
add(label1);
add(label2);
add(label3);
}
}

```

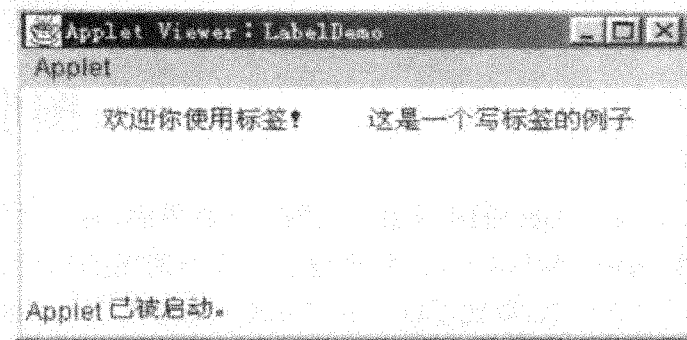


图 8.1

## 2. 标签的构造方法

- (1) `Label()` 用来创建一个没有显示内容的对象。
- (2) `Label(String label)` 用来创建一个显示内容为 `label` 的对象。
- (3) `Label(String label, int alignment)` 除了用来创建一个显示内容为 `label` 的对象外,还设置了 `Label` 的对齐方式。

`Label` 的对齐方式有三种,分别用 `Label` 类的三个常量 `LEFT`、`CENTER` 和 `RIGHT` 来表示左对齐、居中对齐和右对齐。

## 3. 标签的常用方法

- (1) `public int getAlignment()` 返回当前的对齐方式。
- (2) `public String getText()` 返回当前显示的字符串。
- (3) `public void setAlignment(int alignment)` 设置对齐方式。
- (4) `public void setText(String label)` 设置显示的字符串。

### 8.1.2 按钮

Java 提供了标准按钮(`Button`),可带有文字标题。

#### 1. 创建按钮

**例 8.2** 创建按钮的例子,如图 8.2。

```

import java.awt.*;
import java.applet.Applet

```

```

public class ButtonDemo extends Applet {
    Button button=new Button("确定");
    public void init() {
        add (button);
    }
}

```

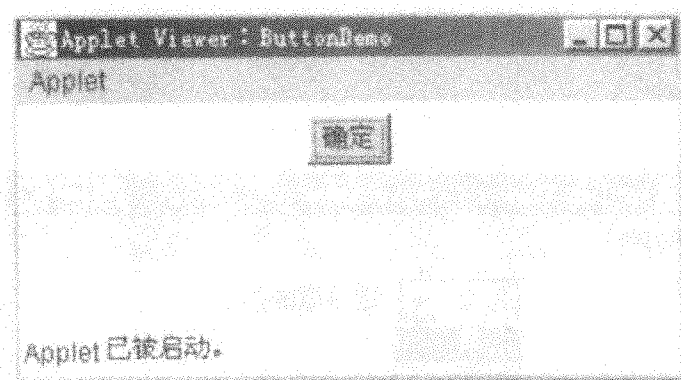


图 8.2

在例 8.2 中,button 是作为主类的一个对象成员创建的,因此,主类的所有方法都可以使用它。参数“确定”指定了按钮上显示的标题。

## 2. 按钮的构造方法

- (1) Button() 创建一个没有标题的按钮。
- (2) Button(String label) 创建一个有显示标题的按钮。

## 3. 按钮的常用方法

- (1) public String getLabel() 返回按钮的显示标题。
- (2) public void setLabel(String label) 设置按钮上的显示标题。

### 8.1.3 选项框

选项框(Choice)又称下拉式列表。这种选项框一次只能显示一个选项,要改变被选中的选项,可以单击下箭头,从选项框中选择一个选项。

#### 1. 创建选项框

**例 8.3** 创建一个选项框的实例,如图 8.3。

```

import java.awt.*;
import java.applet.*;
public class ChoiceDemo extends Applet {
    public void init() {
        Label label=new Label("选项框的例子");
        Choice c=new Choice();
    }
}

```

```

        c.addItem("北京");
        c.addItem("上海");
        c.addItem("天津");
        c.addItem("南京");
        c.addItem("郑州");
        c.addItem("武汉");
        add(c);
        add(label);
    }
}

```

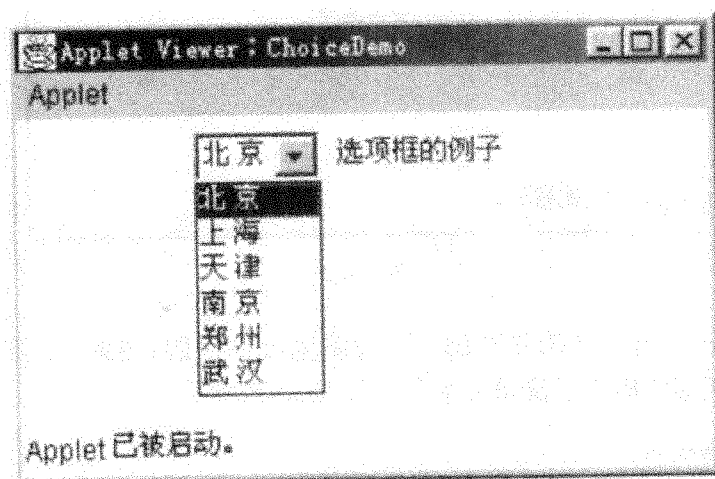


图 8.3

## 2. 选项框的常用方法

- (1) `public void addItem(String item)` 向选项框中加入选项 item。
- (2) `public int countItem()` 返回选项框中的选项个数。
- (3) `public String getItem(int index)` 返回指定下标值的某个选项。
- (4) `public int getSelectIndex()` 返回被选中的选项的下标值。
- (5) `public String getSelectItem()` 返回被选中的选项。
- (6) `public void select(int pos)` 选择指定下标值的选项。
- (7) `public void select(String str)` 选择指定的选项。

### 8.1.4 复选框和选项按钮

复选框(Checkbox)可以让用户作出多项选择。选项按钮(CheckboxGroup)又称单选框,是一组按钮,用户只能选择其中的一个。

#### 1. 创建复选框

例 8.4 创建复选框的例子,如图 8.4。



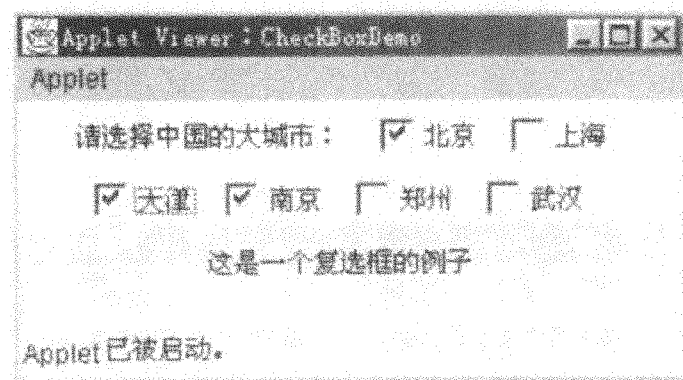


图 8.4

```
import java.awt.* ;
import java.applet. Applet;
public class CheckboxDemo extends Applet {
    final static int CITY_NUM=6;
    String city[]={"北京","上海","天津","南京","郑州","武汉"};
    Checkbox c[]=new Checkbox[6];
    Label label=new Label("这是一个复选框的例子");

    public void init() {
        add(new Label("请选择中国的大城市:"));
        for (int i=0; i<CITY_NUM; i++) {
            c[i]=new Checkbox(city[i]);
            add(c[i]);
        }
        add(label);
    }
}
```

**例 8.5** 创建选项按钮的例子,如图 8.5。

```
import java.awt.* ;
import java.applet. Applet;
public class CheckboxGroupDemo extends Applet {
    final static int CITY_NUM=6;
    String city[]={"北京","上海","天津","南京","郑州","武汉"};
    Checkbox radio[]=new Checkbox[6];
    Label label=new Label("这是一个选项按钮的例子");

    public void init() {
        CheckboxGroup c=new CheckboxGroup();
        add(new Label("请选择中国最大的城市:"));
        for (int i=0; i<CITY_NUM; i++) {
            radio[i]=new Checkbox(city[i], c, false);
        }
    }
}
```

```

        add(radio[i]);
    }
    add(label);
}
}

```

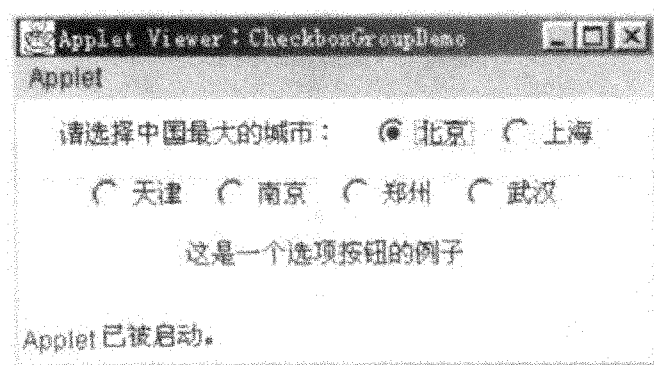


图 8.5

## 2. 复选框和选项按钮的构造方法

(1) `Checkbox()` 创建一个没有标签的复选框。

(2) `Checkbox("北京")` 创建一个有标签的复选框。

(3) `CheckboxGroup()` 创建选项按钮。

要生成选项按钮,必须先生成一个类 `CheckboxGroup` 的对象,例如:

```
checkboxGroup1 = new CheckboxGroup();
```

然后再使用下面的语句:

```
checkbox1 = new Checkbox("Radio", checkboxGroup1, false);
```

其中第一个参数是选项按钮的标签,第二个参数是选项按钮对象,第三个参数表示该选项按钮创建时,是否被选中(true 表示选中)。

## 3. 复选框和选项按钮的常用方法

(1) `public CheckboxGroup getCheckboxGroup()` 返回选项按钮所属的复选框组。

(2) `public String getLabel()` 返回复选框或选项按钮的标签。

(3) `public Boolean getState()` 返回复选框或选项按钮是否被选中。

(4) `public void setCheckboxGroup(CheckboxGroup g)` 设置选项按钮所属复选框组。

(5) `public void setLabel(String label)` 设置复选框或选项按钮的标签。

(6) `public void setState(Boolean state)` 设置复选框或选项按钮被选中。

### 8.1.5 列表框

列表框(List)可以使用户选择多个选项。列表框的所有选项都是可见的,如果选项

数目超出了列表框可见区的范围,则列表框右边会出现一个滚动条。

例 8.6 创建列表框、添加选项,如图 8.6。

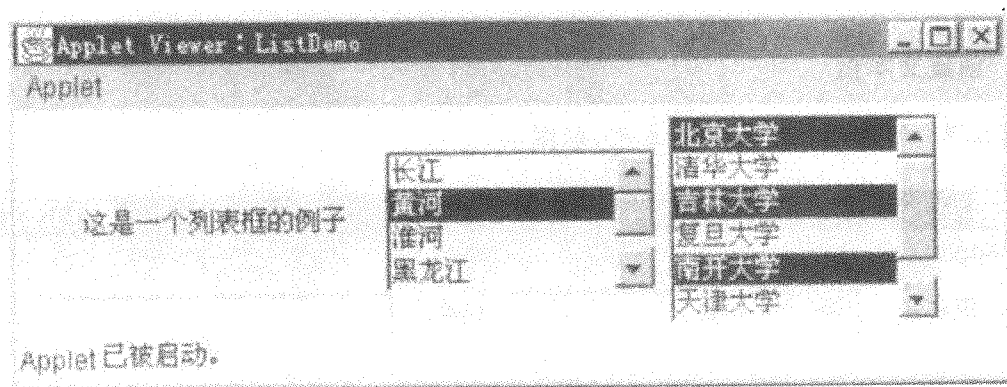


图 8.6

```
import java.awt.* ;
import java.applet.Applet;
public class ListDemo extends Applet {
    List list1 = new List();
    List list2 = new List(6, true);

    public void init() {
        add(new Label("这是一个列表框的例子"));
        list1.add("长江");
        list1.add("黄河");
        list1.add("淮河");
        list1.add("黑龙江");
        list1.add("金沙江");
        add(list1);
        list2.add("北京大学");
        list2.add("清华大学");
        list2.add("吉林大学");
        list2.add("复旦大学");
        list2.add("南开大学");
        list2.add("天津大学");
        list2.add("南京大学");
        add(list2);
    }
}
```

说明:构造方法中的第一个参数指定了在列表框里显示选项的个数。第二个参数若为“false”,表示这个列表框是单选的,若为“true”,则表示是多选的。无参构造方法将创建一个默认大小的列表框。

### 8.1.6 文本框

文本框(TextField)用来接受用户键盘输入的单行文本信息。

#### 1. 创建文本框

例 8.7 创建两个不同的文本框,如图 8.7。

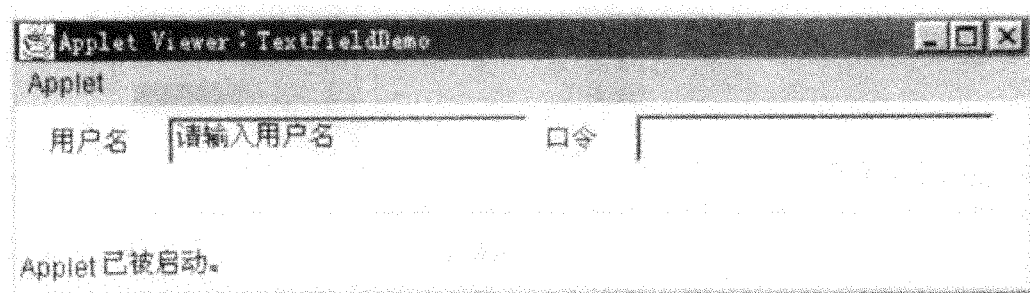


图 8.7

```
import java.awt.*;  
import java.applet.Applet;  
public class TextField Demo extends Applet {  
    public void init() {  
        add(new Label("用户名"));  
        add(new TextField("请输入用户名", 20));  
        add(new Label("口令"));  
        add(new TextField(20));  
    }  
}
```

#### 2. 文本框的构造方法

- (1) TextField() 创建一个默认长度的文本框。
- (2) TextField(int columns) 创建一个指定长度的文本框。
- (3) TextField(String text) 创建一个带有初始文本内容的文本框。
- (4) TextField(String text, int columns) 创建一个带有初始文本内容并具有指定长度的文本框。

#### 3. 文本框的常用方法

- (1) public void setEchoChar(char c) 设定用户键入字符的回显字符,例如输入口令时可设定回显字符“\*”来屏蔽。
- (2) public void setText(String t) 设定文本框的文本内容。

### 8.1.7 文本区

与文本框只显示一行不同,文本区(TextArea)可以显示大段的文本。

## 1. 创建文本区

例 8.8 创建两个不同的文本区,如图 8.8。

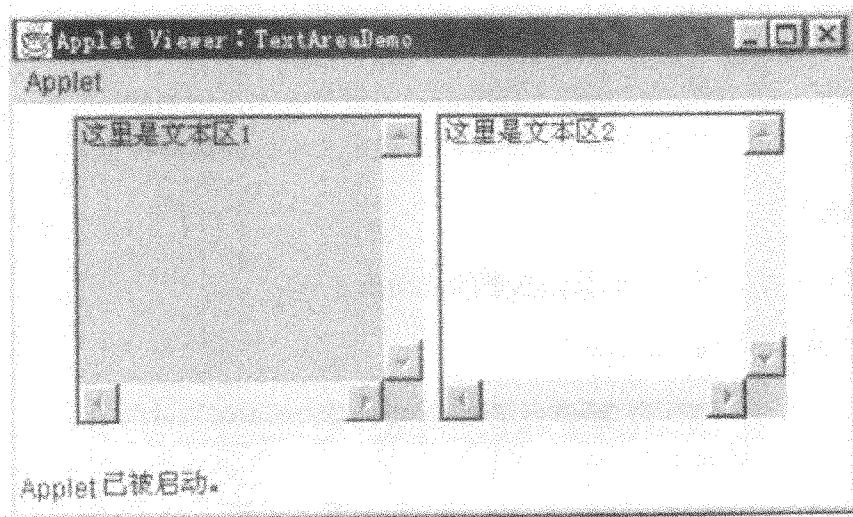


图 8.8

```
import java.awt.* ;
import java.applet.* Applet;
public class TextAreaDemo extends Applet {
    public void init() {
        TextArea ta1 = new TextArea("这里是文本区 1",8,20);
        add(ta1);
        ta1.setEditable(false);
        TextArea ta2 = new TextArea("这里是文本区 2",8,20);
        add(ta2);
        ta2.setEditable(true);
    }
}
```

## 2. 文本区的构造方法

- (1) `TextArea()` 创建一个默认大小的文本区。
- (2) `TextArea(int rows, int columns)` 创建一个指定行和列数的文本区。
- (3) `TextArea(String text)` 创建一个带有初始文本内容的文本区。
- (4) `TextArea(String text, int rows, int columns)` 创建一个带有初始文本内容并具有指定行和列数的文本区。
- (5) `TextArea(String text, int rows, int columns, int scrollbars)` 在(4)的基础上添加滚动条。

## 3. 文本区的常用方法

- (1) `public void append(String str)` 在文本区尾部添加文本。

- (2) `public void insert(String str, int pos)` 在文本区指定位置插入文本。
- (3) `public void setText(String t)` 设定文本区内容。
- (4) `public int getRows()` 返回文本区的行数。
- (5) `public void setRows(int rows)` 设定文本区的行数。
- (6) `public int getColumns()` 返回文本区的列数。
- (7) `public void setColumns(int columns)` 设定文本区的列数。
- (8) `public void setEditable(boolean b)` 设定文本区的读写状态。

### 8.1.8 滚动条

类 `Scrollbar` 可以用来产生我们熟悉的滚动条。

例 8.9 创建滚动条,如图 8.9。

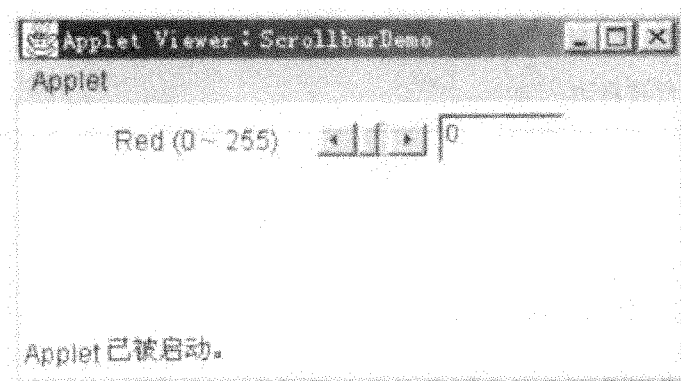


图 8.9

```
import java.awt.*;
import java.applet.Applet;
public class ScrollbarDemo extends Applet {
    Scrollbar s;
    TextField t;
    public void init() {
        s=new Scrollbar(Scrollbar.HORIZONTAL,0,50,0,255);
        t=new TextField("0",5);
        t.setEditable(true);
        add(new Label("Red (0~255)"));
        add(s);
        add(t);
    }
}
```

程序中定义了一个水平方向、初始值为 0、可见范围的宽度为 50、最小值和最大值分别为 0 和 255 的滚动条。

创建一个滚动条时,必须指定它的方向(垂直 `VERTICAL` 或水平 `HORIZONTAL`,其中 `VERTICAL` 是默认值)、初始值、可见范围的宽度、最小值和最大值。滚动条的当前

值可用 `getValue()` 获取,该方法返回一个整型数。

## 8.2 组件布局管理

在上面的例子中你会发现,组件的位置由容器的默认布局管理器摆放,这样当组件较多时窗口就会显得凌乱。如何控制组件的摆放位置呢?本节介绍的布局管理器就是解决这个问题的,它可以实现 5 种布局方式。

### 8.2.1 顺序布局

顺序布局(`FlowLayout`)是最基本的一种布局,是面板(`Panel`)和它的子类 `Applet` 的默认布局方式,前面介绍的例子都使用了默认的顺序布局。`Panel` 是一个容器,可以容纳多个组件,作为它的子类,`Applet` 也具有容纳组件的能力。

顺序布局指的是把组件一个接一个地从左到右顺序排列,一行排满后就转到下一行继续排列,直到把所有组件都显示出来。

在顺序布局方式下,一个组件使用容器的 `add` 方法就可以把自己加入到容器的组件队列中。由于顺序布局功能有限,不能很好地控制组件的排列,所以常用在组件较少的情况下,组件较多时可采用其他布局方式。可使用容器的 `setLayout` 方法改变组件布局方式。

### 8.2.2 边界布局

边界布局(`BorderLayout`)把容器(这里是 `Applet`)分为 5 个区:北区、南区、东区、西区和中区。这几个区的分布规律是“上北下南,左西右东”,与地图的方位相同。组件可以指定自己放在那个区内,因为只有 5 个区,所以最多只能容纳 5 个组件,否则要采取其他布局方式。下面是一个边界布局的例子:

例 8.10 使用边界布局添加 5 个组件,如图 8.10。

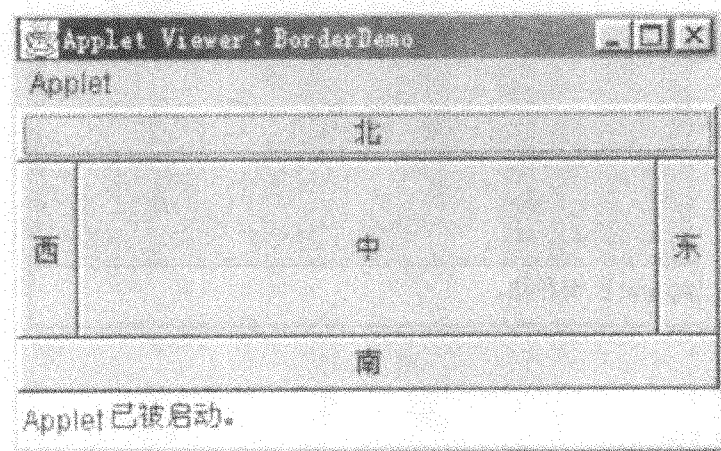


图 8.10

```

import java.awt. * ;
import java.applet. Applet;
public class BorderDemo extends Applet {
    Button bN, bS, bW, bE, bC;
    public void init() {
        setLayout(new BorderLayout());
        bN=new Button("北");
        bS=new Button("南");
        bE=new Button("东");
        bW=new Button("西");
        bC=new Button("中");
        add("North", bN);
        add("South", bS);
        add("East", bE);
        add("West", bW);
        add("Center", bC);
    }
}

```

说明:在 add 方法中,第一个参数表示组件的摆放位置,必须从 North、South、East、West、Center 中选择一个。

### 8.2.3 卡片布局

卡片布局(CardLayout)将组件像卡片一样叠放起来,每次只显示一个。因此你需要使用某种方法翻阅这些卡片,看看下面的例子。

例 8.11 组件的卡片布局,如图 8.11。

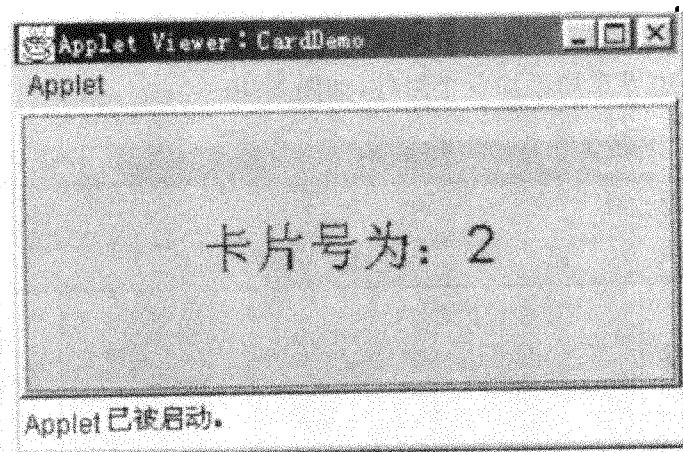


图 8.11

```

import java.awt. * ;
import java.applet. Applet;
public class CardDemo extends Applet {

```



```

CardLayout card=new CardLayout();
public void init() {
    setLayout(card);
    setFont(new Font("Arial", Font.PLAIN, 24));
    for (int i=1; i<=5; i++) {
        add(String.valueOf(i), new Button("卡片号为:"+i));
    }
    card.show (this, String.valueOf(2));
}
}

```

说明:使用卡片布局时,首先要创建一个卡片布局管理器对象,例如程序中创建的 card 对象。然后用 setLayout(card) 设定容器的布局方式。由于布局管理器将组件叠放起来时要确定组件的编号,这可通过 add(字符串编号, 组件) 方法来加入组件。显示一个组件时要通过布局管理器的 show 方法,它需要两个参数:容器对象名和组件的字符串编号。

其他卡片被当前卡片覆盖了,需要使用事件处理程序才可以翻开,事件处理程序我们在下面介绍。程序中还使用了设置字体的方法 setFont,在第 10 章中有介绍。

#### 8.2.4 网格布局

网格布局(GridLayout)把容器(这里是 Applet)区域分成若干个网格,每个网格可以放置一个组件,这种布局方式对数量众多的组件很合适。创建网格布局管理器时,可以给出网格的行数和列数。我们看下面的例子。

例 8.12 组件的网格布局,如图 8.12。

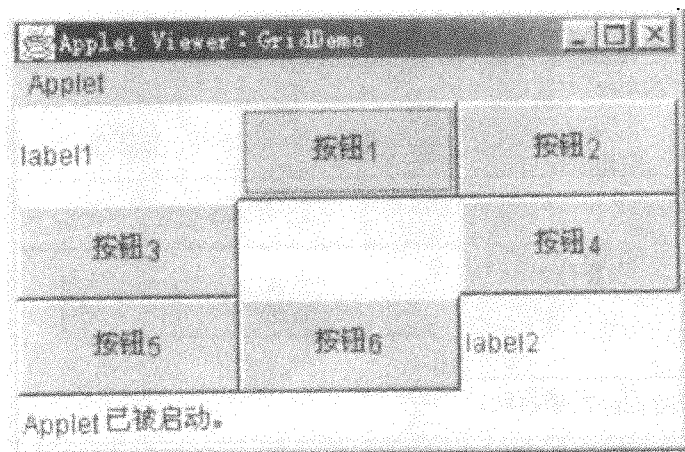


图 8.12

```

import java.awt.*;
import java.applet.*;
public class GridDemo extends Applet {
    Button b1, b2, b3, b4, b5, b6;

```

```

public void init() {
    setLayout(new GridLayout(3,3)); // 设置为 3 行 3 列共 9 个网格
    b1 = new Button("按钮 1");
    b2 = new Button("按钮 2");
    b3 = new Button("按钮 3");
    b4 = new Button("按钮 4");
    b5 = new Button("按钮 5");
    b6 = new Button("按钮 6");
    add(new Label("label1"));
    add(b1); add(b2); add(b3);
    add(new Label());
    add(b4); add(b5); add(b6);
    add(new Label("label2"));
}
}

```

说明：程序在设置网格布局的同时定义了网格为 3 行 3 列。网格数可以比组件多，但不能少。如果希望某个网格为空白，可以为它加上一个空标签。在网格布局中也可以添加间距，如 `setLayout(new GridLayout(3, 3, 10, 10))` 将网格之间设为 10 个点距。

### 8.2.5 网格包布局

网格布局中所有组件被布局管理器设为默认大小，且大小相同，因此界面设计出来并不理想。使用网格包布局既可以实现网格布局的效果，又可以使组件具有不同的大小。网格包布局是最灵活的但也是最复杂、最难掌握的布局管理器。

例 8.13 网格包布局应用实例，如图 8.13。

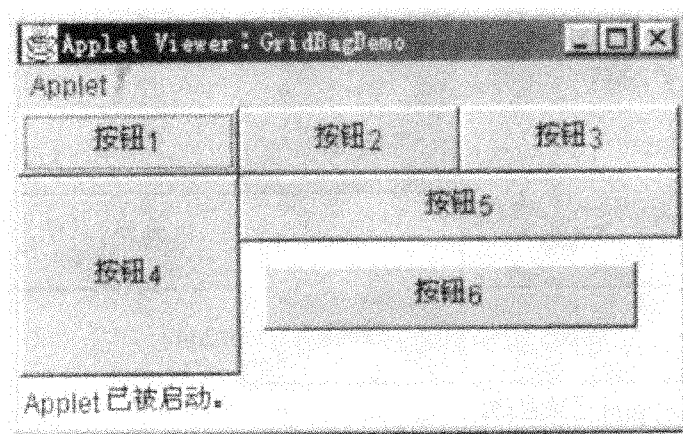


图 8.13

```

import java.awt.*;
import java.applet.*;
public class GridBagDemo extends Applet {
    GridBagLayout gb = new GridBagLayout();
    GridBagConstraints gbc = new GridBagConstraints();

```

```

void AddNewButton(String BtnName) {
    Button btn=new Button(BtnName);
    gbc.setConstraints(btn,gbc); // 设定组件的位置和大小后才能加入组件
    add(btn);
}

public void init() {
    setLayout(gbc);
    gbc.weightx=1.0; // 所有组件平均分配容器空间
    gbc.weighty=1.0;
    gbc.fill=gbc.BOTH; // 所有组件都会把分配到的空间填满

    AddNewButton("按钮 1"); // 按钮 1、按钮 2 和按钮 3 位于第 1 行
    AddNewButton("按钮 2");
    gbc.gridwidth=gbc.REMAINDER; // 剩余空间分配给按钮 3
    AddNewButton("按钮 3");

    gbc.gridwidth=1; // 按钮 4 在横向上占 1 个方格
    gbc.gridheight=2; // 按钮 4 在纵向上占 2 个方格
    AddNewButton("按钮 4"); // 按钮 4、按钮 5 和按钮 6 位于第 2 行
    gbc.gridwidth=gbc.REMAINDER; // 剩余空间分配给按钮 5 和按钮 6
    gbc.gridheight=1; // 按钮 5 在纵向上占 1 个方格
    AddNewButton("按钮 5");

    Insets OrigInsets=gbc.insets;
    gbc.insets=new Insets(10,10,20,20); // 设定按钮 6 和其他按钮的间距
    AddNewButton("按钮 6");
    gbc.insets=OrigInsets;
}
}

```

在网格包布局中,每个组件首先要由 GridBagConstraints 类的对象来设定属性,如组件的位置和大小等,然后才能加入到容器中。程序中用一个方法 AddNewButton 来创建和添加按钮组件。在这个方法中,先创建按钮,然后调用布局管理器的 setConstraints 设定按钮的限制条件(已在 init 方法中确定),最后调用容器的 add 方法加入按钮。

在 init 方法中,首先设定网格包布局方式,然后为每一个按钮分配容器空间。这样按钮的位置和大小就确定下来了,可以调用 AddNewButton 方法添加这个按钮。对每一个按钮重复这个过程,直到按钮分配完毕。

GridBagConstraints 的限制条件主要有:

(1) gridwidth 和 gridheight

指定组件在横向和纵向上占有的网格数。设为常量 REMAINDER 表示把横向和纵向剩下的方格都分配给该组件。

### (2) gridx 和 gridy

指定组件摆放的网格位置。默认值为 RELATIVE, 表示该组件紧接着上一个摆放, 二者均为 0 表示放在左上角。

### (3) weightx 和 weighty

表示当 GridBagLayout 按照 GridBagConstraints 所设置的限制条件安排好以后, 如果容器中仍有剩余空间, 为该组件分配的比例数。二者均为 0, 表示不能被分配到剩余空间。

### (4) fill

如果组件分配的空间大于它所需要的空间时, GridBagLayout 会根据 fill 的值来调整该组件的大小。NONE 表示不调整(默认值), BOTH 表示在水平和垂直方向上都调整。

### (5) insets

insets 属于 Insets 类型, 这个属性表明组件与所分配空间四边的间距。默认为 (0, 0, 0, 0), 表示与上、左、右、下四边的间距为 0。

## 8.2.6 面板的使用

面板 (Panel) 是一个无边框的容器, 可以包容其他组件或另一个面板。使用面板的目的是为了分层次、分区域管理各种组件, 通过各个面板的布局管理器对本身的组件进行管理, 互不妨碍, 这样就可以使布局更加合理和美观。

在前面的例子里我们没有使用面板, Applet 本身就是一个特殊的面板, 现在让我们看看如何在 Applet 中添加面板进行组件布局。

**例 8.14** 用面板控制组件布局, 如图 8.14。

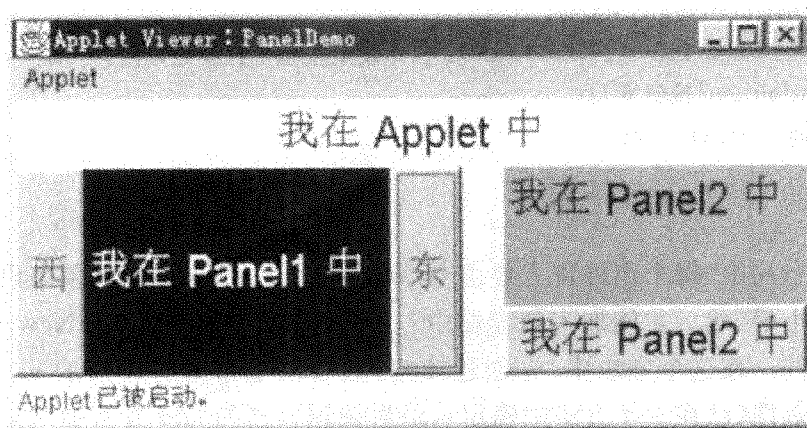


图 8.14

```
import java.awt.*;  
import java.applet.Applet;  
public class PanelDemo extends Applet {  
    public void init() {  
        setLayout(new BorderLayout());  
        setForeground(Color.black);  
    }  
}
```

```

setFont(new Font("Arial",Font.PLAIN,20));
add("North", new Label("我在 Applet 中",Label.CENTER));

Panel panel1 = new Panel();
add("West", panel1);
panel1.setBackground(Color.blue);
panel1.setForeground(Color.red);
panel1.setLayout(new BorderLayout());
panel1.add("East", new Button("东"));
panel1.add("West", new Button("西"));
panel1.add("Center", new Label("我在 Panel1 中"));

Panel panel2 = new Panel();
add("East", panel2);
panel2.setBackground(Color.green);
panel2.setLayout(new BorderLayout());
panel2.add("North", new Label("我在 Panel2 中"));
panel2.add("South", new Button("我在 Panel2 中"));
}
}

```

程序中以 Applet 为容器,加入了两个面板 panel1 和 panel2。作为容器,三者的功能是一样的,都可以包容其他组件。这里,Applet 是最底层的容器,panel1 和 panel2 依附在它的上边。

下面我们给出面板和组件的常用方法:

- Panel() 用默认布局方式创建一个面板。
- Panel(LayoutManager layout) 用指定布局方式创建一个面板。
- public Component add(Component comp) 为容器添加一个组件。
- public Component add(Component comp, int index) 将组件添加到队列的指定位置。
- public void add(Component comp, Object constraints) 按照限制条件添加组件。
- public void remove(Component comp) 去除指定组件。
- public void setFont(Font f) 设定组件的字体。
- public void setSize(Dimension d) 设定组件的宽和高。
- public void setVisible(boolean b) 设定组件可见。
- public void setLocation(int x, int y) 设定组件的位置。
- public void setBackground(Color c) 设定组件的背景色。
- public void setForeground(Color c) 设定组件的前景色。
- public void setBounds(int x, int y, int width, int height) 设定组件的位置和大小。
- public void setName(String name) 设定组件的名称。

- `public String getName()` 返回组件的名称。
- `public int getX()` 返回组件的 x 坐标。
- `public int getY()` 返回组件的 y 坐标。
- `public int getHeight()` 返回组件的高。
- `public int getWidth()` 返回组件的宽。
- `public void paint(Graphics g)` 画出容器。
- `public void update(Graphics g)` 用背景色清除组件, 设定画笔为前景色, 调用 `paint` 方法重画组件。
- `public void repaint()` 立即调用组件的 `update` 方法。

### 8.2.7 手工布局

Java 允许程序员不使用布局管理器, 但此时必须手工放置各个组件。如下面的例子:

**例 8.15** 手工放置组件, 如图 8.15。

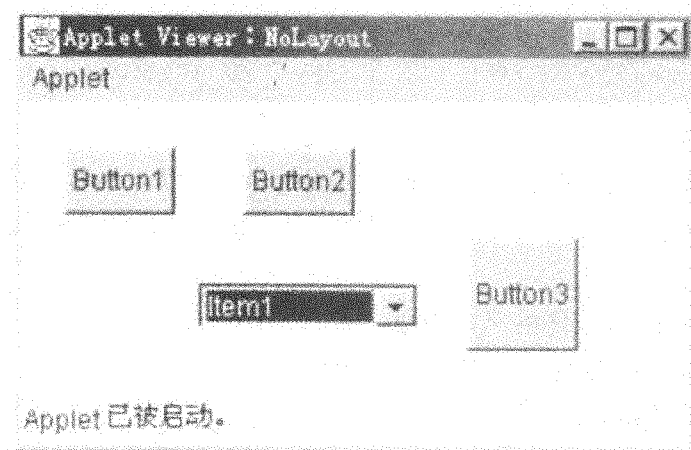


图 8.15

```
import java.awt.*;
import java.applet.Applet;
public class NoLayout extends Applet {
    Choice c=new Choice ();
    Button b1=new Button("Button1");
    Button b2=new Button("Button2");
    Button b3=new Button("Button3");

    public void init() {
        setLayout(null);
        c.addItem("Item1");
        c.addItem("Item2");
        c.addItem("Item3");
        add(c);
```

```

        c.setBounds(80,80,100,20);
        add(b1);
        b1.setBounds(20,20,50,30);
        add(b2);
        b2.setBounds(100,20,50,30);
        add(b3);
        b3.setBounds(200,60,50,50);
    }
}

```

程序中通过 `setLayout(null)` 语句关闭了默认的布局管理器,由程序员自己决定组件的摆放位置和大小。`setBounds` 方法可实现这个功能。该方法有 4 个参数,其中 `x` 指定了组件的水平位置坐标;`y` 指定了组件的纵向位置坐标;`width` 指定了组件的宽度;`height` 指定了组件的高度。例如 `b2.setBounds(100,20,50,30)` 将按钮 `b2` 放在(120, 20)位置,宽度和高度分别是 50 和 30。

## 8.3 事件处理

在 Java 中,程序和用户的交互是通过响应各种事件来实现的。例如,用户点击了一个按钮,意味着一个按钮事件的发生;选中了一个选项,意味着一个选项事件的发生。每当一个事件发生,Java 虚拟机就会将事件的消息传递给程序,由程序中的事件处理方法对事件进行处理。如果没有编写事件处理方法,程序就不能和用户交互。

Java 的事件处理机制和其他 OOP 语言不同,它采用了委托型的事件处理机制,下面就讨论这个问题。

### 8.3.1 Java 的事件处理机制

你也许要奇怪,前面编了这么多 Java 程序,可对按钮做了许多操作,怎么都没有反应呢?目前还不会有反应,因为 Java 虚拟机虽然通知程序发生了按钮点击事件,但程序中并没有相应的事件处理方法,所以事件被忽略了。如果希望能对各种事件作出反应,我们可以编写一个或多个事件处理方法,当程序监听到事件发生后,就可以调用事件处理方法来响应。

在 GUI 中,用户的输入或命令是通过键盘和鼠标来完成的,系统必须能够识别这些事件,然后才能正确处理这些事件。Java 有很多预定义事件类,它们包含了所有组件上可能发生的事件。程序员只需要确定哪个组件将发生什么预定义事件,然后编写针对这个事件的处理方法就可以了。当事件发生后,系统就会自动调用事件处理方法来响应。

很多 OOP 语言是将事件处理方法作为对象的成员方法,发生事件时,由对象自己调用相应的事件处理方法,如 VB、Delphi 等。而 Java 采用了委托型事件处理模式,即对象(指组件)本身没有用成员方法来处理事件,而是将事件委托给事件监听者处理,这就使得组件更加简练。

原书缺页



### 8.3.2 事件处理实例

#### 1. 按钮动作事件处理

例 8.16 以发出声音来响应按钮点击事件,如图 8.16。

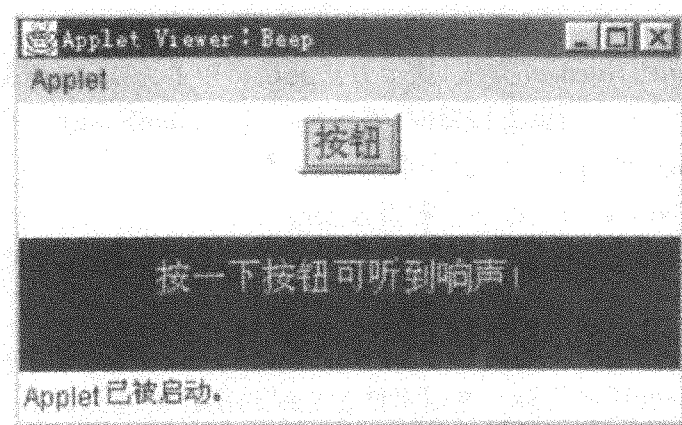


图 8.16

```
import java.awt.*;  
import java.awt.event.*;  
import java.applet.Applet;  
  
public class Beep extends Applet implements ActionListener { // 实现动作事件监听接口  
    public void init() {  
        setLayout(new GridLayout(2,1));  
        setFont(new Font("Arial",Font.PLAIN,16));  
        Panel p1=new Panel();  
        add("Center",p1);  
        Button btn=new Button("按钮");  
        p1.add(btn);  
        btn.addActionListener(this); // 注册事件源的动作监听者  
        Panel p2=new Panel();  
        add(p2);  
        p2.setBackground(Color.blue);  
        p2.setForeground(Color.yellow);  
        p2.add(new Label("按一下按钮可听到响声!", Label.CENTER));  
    }  
  
    public void actionPerformed(ActionEvent e) {  
        Toolkit.getDefaultToolkit().beep(); // 动作事件发生时要作出的反应  
    }  
}
```

说明:程序分别在两个面板上创建了一个按钮和一个标签。用鼠标单击按钮时,会听

到一声响声。

从上例中我们概括出建立动作事件监听器的步骤：

- (1) 引入系统事件类包,如 `import java.awt.event.*`。
- (2) 在定义类的同时声明实现动作事件监听器接口,如 `implements ActionListener`。
- (3) 在方法中调用事件源对象的 `addActionListener`,注册事件源对象的事件监听者,如 `btn.addActionListener(this)`。
- (4) 实现 `actionPerformed` 方法,这个方法是接口中的方法,应该覆盖这个方法,加入代码以响应事件的发生。如单击按钮时,系统将产生 `ActionEvent` 事件,动作事件监听者就调用 `actionPerformed` 方法处理这个事件。

动作事件类 `ActionEvent` 有一些常用方法：

- (1) `public String getActionCommand()` 可返回事件源的标签,如 `btn` 的标签是“按钮”。
- (2) `public Object getSource()` 返回产生事件的对象引用,如 `btn`。
- (3) `public int getModifiers` 返回事件发生时功能控制键的状态,它们可能是功能键常数 `SHIFT_MASK`、`CTRL_MASK`、`ALT_MASK`。如果 `getModifiers` 的返回值不等于这些常数中的任何一个,说明发生事件时没有按住功能控制键不放。

## 2. 选项事件处理

例 8.17 复选框和选项按钮的事件处理,如图 8.17。

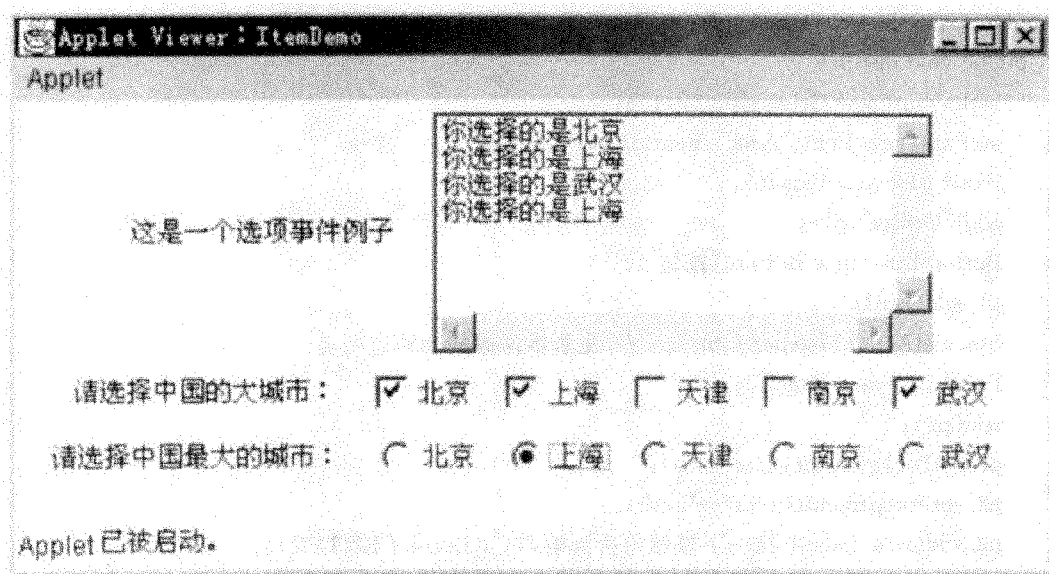


图 8.17

```
import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

public class ItemDemo extends Applet implements ItemListener {
    TextArea area=new TextArea(6,30);
```

```

String City[] = {"北京", "上海", "天津", "南京", "武汉"};
Checkbox cb[] = new Checkbox[5];
Checkbox radio[] = new Checkbox[5];

public void init() {
    add(new Label("这是一个选项事件例子"));
    add(area);
    add(new Label("  请选择中国的大城市:"));
    for(int i=0; i<5; i++) {
        cb[i] = new Checkbox(City[i]);
        add(cb[i]);
        cb[i]. addItemListener(this);
    }
    CheckboxGroup cbGroup = new CheckboxGroup();
    add(new Label("请选择中国最大的城市:"));
    for(int i=0; i<5; i++) {
        radio[i] = new Checkbox(City[i], cbGroup, false);
        add(radio[i]);
        radio[i]. addItemListener(this);
    }
}

public void itemStateChanged(ItemEvent e) {
    area.append("你选择的是" + e.getItem() + "\n");
}
}

```

例 8.18 列表框的事件处理,如图 8.18。

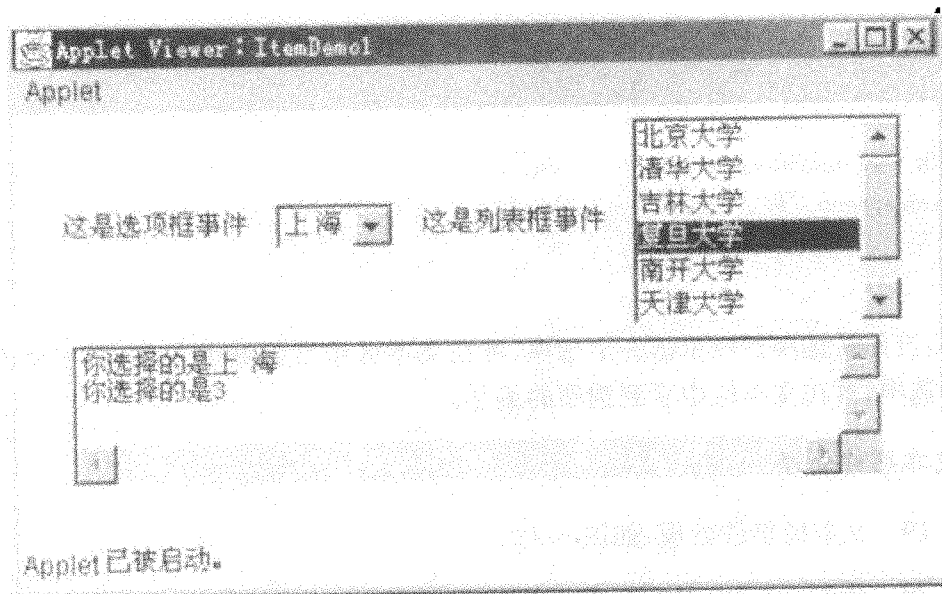


图 8.18

```

import java.applet. Applet;
import java.awt. * ;
import java.awt. event. * ;
public class ItemDemo1 extends Applet implements ItemListener {
    TextArea area=new TextArea(3,50);
    Choice c=new Choice();
    List l=new List(6,true);
    public void init() {
        add(new Label("这是选项框事件",Label. CENTER));
        c.add("北京");
        c.add("上海");
        c.add("天津");
        c.add("南京");
        c.add("郑州");
        c.add("武汉");
        add(c);
        c.addItemListener (this);
        add(new Label("这是列表框事件",Label. CENTER));
        l.add("北京大学");
        l.add("清华大学");
        l.add("吉林大学");
        l.add("复旦大学");
        l.add("南开大学");
        l.add("天津大学");
        l.add("南京大学");
        add(l);
        l.addItemListener (this);
        add(area);
    }

    public void itemStateChanged(ItemEvent e) {
        area.append ("你选择的是"+e.getItem ()+"\n");
    }
}

```

说明：当你在选项框中单击一个选项，可在文本区中看到城市名称；当你在列表框中单击一个选项，可在文本区中看到选项的编号。

### 3. 文本事件处理

**例 8.19** 文本的事件处理，如图 8.19。

```

import java.applet. Applet;
import java.awt. * ;
import java.awt. event. * ;

```

```

public class TextDemo extends Applet implements TextListener {
    TextArea Area=new TextArea (6,40);
    TextField tf1=new TextField("用户名",10);
    TextField tf2=new TextField(10);
    public void init() {
        setFont(new Font("Arial",Font.PLAIN,12));
        add(Area);
        add(new Label("用户名"));
        add(tf1);
        add(new Label("电话"));
        add(tf2);
        tf1.addTextListener(this);
        tf2.addTextListener(this);
    }

    public void textValueChanged(TextEvent e) {
        Area.append ("你好!"+"\n");
    }
}

```

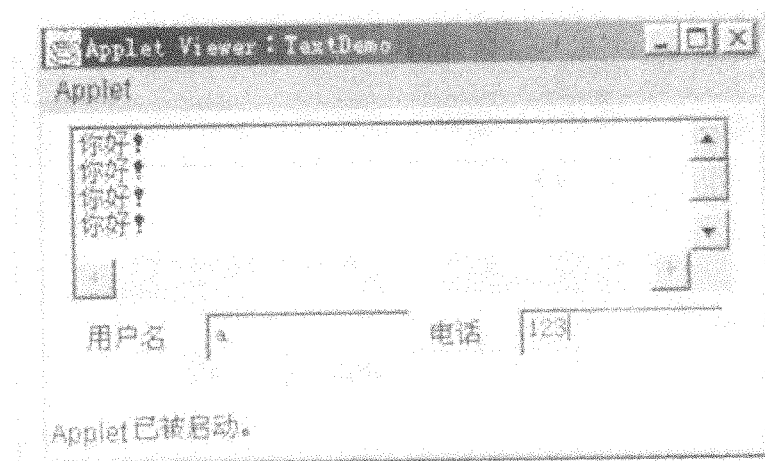


图 8.19

#### 4. 文本动作事件处理

在例 8.19 中,每输入一个字母就激发一个文本改变事件,事件监听者就会调用一次事件处理方法。下面的例子介绍了输入后按回车键的情况,这种情况属于动作事件。

**例 8.20** 文本的动作事件处理,如图 8.20。

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;
public class TextDemol extends Applet implements ActionListener {
    TextArea Area=new TextArea (6,40);

```

```

TextField tf1=new TextField(10);
TextField tf2=new TextField(10);
public void init() {
    setFont(new Font("Arial",Font.PLAIN,12));
    add(Area);
    add(new Label("用户名"));
    add(tf1);
    add(new Label("电话"));
    add(tf2);
    tf1.addActionListener(this);
    tf2.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource()==tf1)
        Area.append("用户名:"+tf1.getText()+"\n");
    else
        Area.append("电话:"+tf2.getText()+"\n");
}
}

```

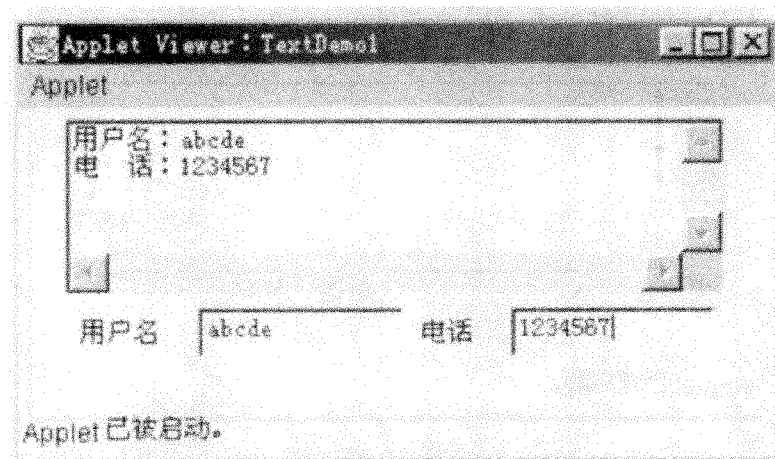


图 8.20

说明:在程序的事件处理方法中,我们用 `e.getSource` 方法来判断事件源是哪一个文本框。该方法的返回值是一个对象引用,如果是 `tf1`,则说明目前的操作是输入用户名,就用 `tf1` 的 `getText` 方法返回用户名并在文本区输出。否则,说明目前的操作是输入电话,就用 `tf2` 的 `getText` 方法返回电话并在文本区输出。

## 5. 调整事件处理

**例 8.21** 显示滚动条滑块的值,如图 8.21。

```
import java.applet.Applet;
```

```

import java.awt. * ;
import java.awt.event. * ;
public class AdjustDemo extends Applet implements AdjustmentListener {
    Scrollbar r;
    TextField t;
    public void init() {
        setLayout(new GridLayout(3,2));
        r=new Scrollbar(Scrollbar.HORIZONTAL,0,10,0,100);
        t=new TextField("0",5);
        t.setEditable(false);
        add(new Label("滚动条值范围(0~100)",Label.CENTER));
        add(r);
        add(t);
        r.addAdjustmentListener(this);
    }

    public void adjustmentValueChanged(AdjustmentEvent e) {
        t.setText(String.valueOf(r.getValue()));
    }
}

```

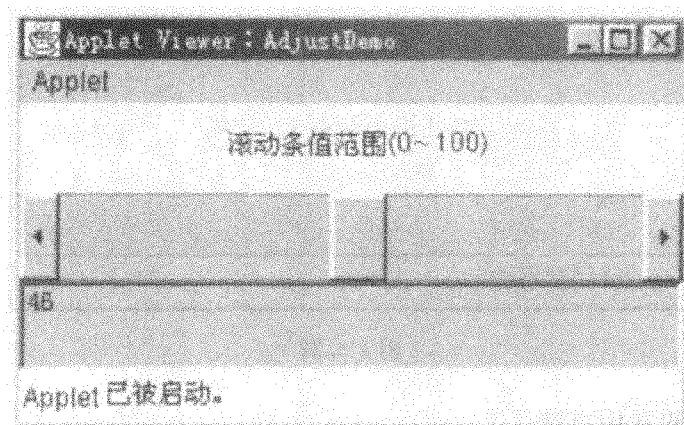


图 8.21

说明：调整事件的处理和其他事件的处理类似，这里使用方法 `t.setText` 重新设置了文本域中显示的内容。滚动条的方法 `r.getValue` 可返回发生事件时滑块的值。String 类的方法 `valueOf` 可将数值转换成字符串。

## 6. 卡片布局的事件处理

**例 8.22** 添加卡片按钮来翻看卡片组件，如图 8.22。

```

import java.applet. Applet;
import java.awt. * ;
import java.awt.event. * ;

```

```

public class CardDemol extends Applet implements ActionListener {
    CardLayout card=new CardLayout();
    public void init() {
        Button cb[]=new Button[6];
        setLayout(card);
        setFont(new Font("Arial",Font.PLAIN,24));
        for (int i=1;i<=5;i++) {
            cb[i]=new Button("卡片号为:"+i);
            add(String.valueOf(i),cb[i]);
            cb[i].addActionListener(this);
        }
        card.show(this,String.valueOf(1));
    }

    public void actionPerformed(ActionEvent e) {
        card.next(this);
    }
}

```

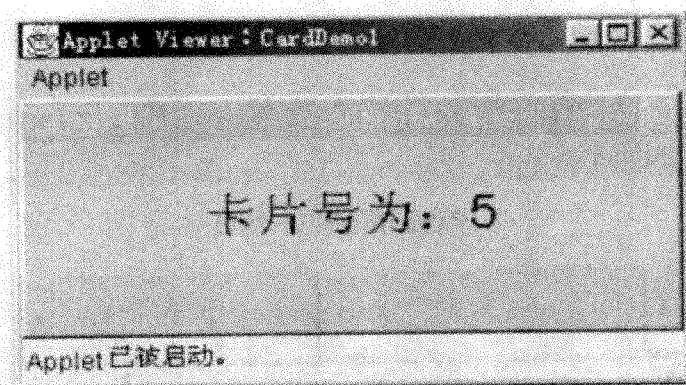


图 8.22

卡片布局提供的常用方法:

- (1) public void first(Container parent) 显示第一张卡片。
- (2) public void last(Container parent) 显示最后一张卡片。
- (3) public void next(Container parent) 显示下一张卡片。
- (4) public void previous(Container parent) 显示上一张卡片。
- (5) public void show(Container parent, String name) 显示指定的卡片。

## 7. 实现多个事件接口

**例 8.23** 在容器上发生的鼠标点击事件和鼠标移动事件,如图 8.23。

```

import java.applet.Applet;
import java.awt.*;
import java.awt.event.*;

```



```

public class Mouse extends Applet implements MouseListener, MouseMotionListener {
    Panel p1 = new Panel();
    Panel p2 = new Panel();
    TextArea area = new TextArea (5,40);

    public void init() {
        setLayout(new BorderLayout());
        add("North", area);
        add("West", p1);
        add("East", p2);
        p1.add(new Label("这是第一个面板", Label.CENTER));
        p2.add(new Label("这是第二个面板", Label.CENTER));
        p1.setBackground(Color.blue);
        p1.addMouseListener(this);
        p1.addMouseMotionListener(this);
        p2.setBackground(Color.green);
        p2.addMouseListener(this);
        p2.addMouseMotionListener(this);
    }

    public void mousePressed(MouseEvent e) {
        if (e.getSource() == p1)
            area.append("你在 Panel1 (" + e.getX() + ", " + e.getY() + ") 按下鼠标\n");
        else
            area.append("你在 Panel2 (" + e.getX() + ", " + e.getY() + ") 按下鼠标\n");
    }

    public void mouseClicked(MouseEvent e) {
        if (e.getSource() == p1)
            area.append("你在 Panel1 (" + e.getX() + ", " + e.getY() + ") 点击鼠标\n");
        else
            area.append("你在 Panel2 (" + e.getX() + ", " + e.getY() + ") 点击鼠标\n");
    }

    public void mouseEntered(MouseEvent e) {
        if (e.getSource() == p1)
            area.append("鼠标进入 Panel1\n");
        else
            area.append("鼠标进入 Panel2\n");
    }

    public void mouseExited(MouseEvent e) {
        if (e.getSource() == p1)
            area.append("鼠标退出 Panel1\n");
        else
            area.append("鼠标退出 Panel2\n");
    }
}

```

```

public void mouseReleased(MouseEvent e) {
    area.append("释放鼠标\n");
}
public void mouseDragged(MouseEvent e) {
    area.append("鼠标拖动 (" + e.getX() + "," + e.getY() + ")\n");
}
public void mouseMoved(MouseEvent e) {
}
}

```

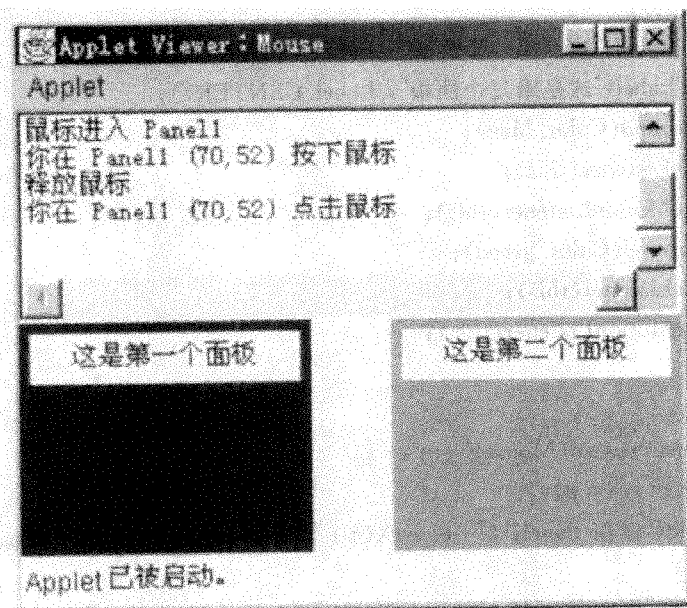


图 8.23

## 习 题

- 8-1 图形用户界面与字符界面有什么不同?
- 8-2 图形用户界面由什么构成? 分析它们的作用。
- 8-3 编写一个 Applet, 添加两个标签、一个文本框、一个文本区和一个按钮。
- 8-4 为第 8-3 题的 Applet 添加事件处理功能。要求在文本框中输入字符串, 当按下回车键或点击按钮时, 可将字符串显示在文本区中。
- 8-5 编写一个 Applet, 添加两个标签、一个选项框、一个列表框。
- 8-6 为第 8-5 题的 Applet 添加事件处理功能。要求在选项框或列表框选择后, 结果被复制到对应的标签中显示出来。
- 8-7 Java 有几种布局方式? 简述常用的布局方式。
- 8-8 使用边界布局在中间放一个文本区, 上面放一个标签, 其他位置放三个按钮。
- 8-9 在面板上使用网格布局放置两个标签、两个文本框、两个按钮。

**8-10** 使用三个面板分别放置两个标签、两个文本框、两个按钮。

**8-11** 编写一个 Applet,添加一个滚动条和一个文本框,移动滚动条的滑块能在文本框中显示出 1 到 1999 的值。

**8-12** 编写一个模拟计算器的 Applet,使用面板和网格布局,添加一个文本框,10 个数字按钮(0~9),4 个加减乘除按钮,一个等号按钮,一个清除按钮。要求将计算公式和结果显示在文本框中。



# 第9章

## 窗口、菜单和对话框

上一章我们介绍了图形用户界面的基本组件,本章将继续介绍图形用户界面的高级组件。这些组件包括窗口、菜单和对话框等,它们可以构造出标准 GUI 应用程序。最后,简要介绍了 Java Swing 的特色。

### 9.1 窗口

Java 的窗口由 Frame 类生成,遗憾的是它不具有关闭功能,必须要在程序中实现窗口事件监听接口并编写关闭窗口的代码才行。本节介绍了创建可关闭窗口的几种方法以及窗口的使用。

#### 9.1.1 创建可关闭窗口

下面是一个应用程序,它弹出一个窗口,可以移动、改变大小、最大化、变成图标,并且可以关闭。

**例 9.1** 方式 1:创建一个可关闭的空白窗口,如图 9.1。

```
import java.awt.*;  
import java.awt.event.*;  
class W1 extends Frame implements WindowListener {  
    W1() {  
        super("Window1");  
        setSize(350,200);  
        setVisible(true);  
        addWindowListener(this);  
    }  
  
    public void windowClosing(WindowEvent e) {  
        System.exit(0);  
    }  
  
    public void windowOpened(WindowEvent e) {}  
}
```

```

public void windowClosed(WindowEvent e) {}
public void windowIconified(WindowEvent e) {}
public void windowDeiconified(WindowEvent e) {}
public void windowActivated(WindowEvent e) {}
public void windowDeactivated(WindowEvent e) {}

public static void main(String args[]) {
    new W1();
}
}

```

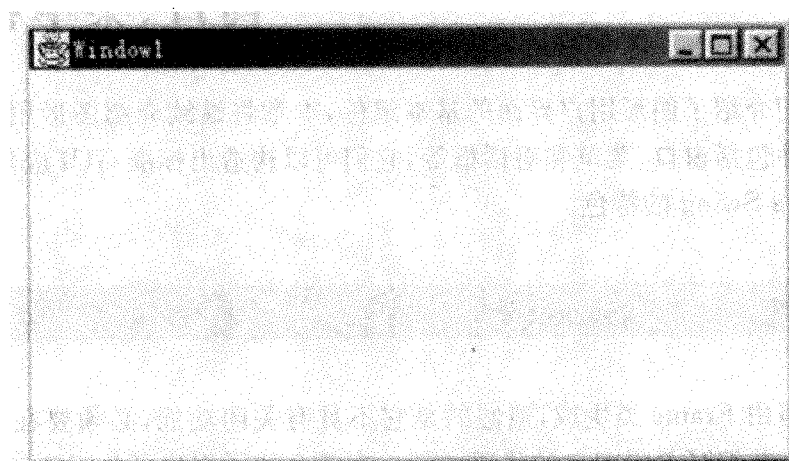


图 9.1

main 方法只有一条语句,生成了 W1 的对象。在 W1 的构造方法中,调用父类的构造方法为窗口指定名称为 Window1,然后指定窗口的宽度和高度分别为 350 和 200,指定窗口可见,最后将窗口对象本身注册为事件监听者。这 4 条语句对于生成一个窗口来说都是必需的。

上例使用了正统的事件监听接口方式为窗口加入关闭方法。但如果仅仅为了增加窗口的关闭功能而采用这种方式则不是最好的选择,因为你必须要实现该接口中的所有抽象方法,不管它们是否用得上。从程序中可以看出,窗口事件监听接口有 7 个方法,有 6 个没有用上,只在 windowClosing 方法中加入了一行代码 System.exit(0) 用来返回操作系统。

为窗口添加关闭功能可以有更简单的方式,通常是利用窗口事件裁剪器,将不需要的事件处理方法裁剪掉,我们在下面的例子里分别介绍了 2 种不同方式来关闭窗口。

**例 9.2** 方式 2:创建一个可关闭的空白窗口。

```

import java.awt.*;
import java.awt.event.*;
public class W2 extends Frame {
    W2() {
        super("Window2");
    }
}

```

```

        setSize(350,200);
        setVisible(true);
        addWindowListener(new Win());
    }

    public static void main(String[] args) {
        new W2();
    }
}

class Win extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}

```

方式 2 使用了窗口事件裁剪器。窗口事件裁剪器 WindowAdapter 是一个系统类,它实现了窗口事件监听接口,覆盖了接口中定义的 7 个方法。裁剪器的作用实际上就是把接口变成一个类,这样你就可以随意使用其中的方法了。

我们创建了一个 WindowAdapter 的子类 Win,根据需要只覆盖了一个 windowClosing 方法,其他方法则不需再考虑。在 W2 的构造方法中,将 Win 对象注册为窗口对象的事件监听者。采用这样的处理方式,程序就简单多了。

**例 9.3 方式 3:创建一个可关闭的空白窗口。**

```

import java.awt.*;
import java.awt.event.*;
public class W3 {
    public static void main(String[] args) {
        Frame f=new Frame("Window3");
        f.setSize(350,200);
        f.setVisible(true);
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }
}

```

方式 3 使用了匿名类,我们在第 5 章里介绍过匿名类的概念。本例和上例的不同就在于窗口事件裁剪器的使用,本例取消了上例中的 Win,直接将这个子类的定义作为参数传递给了窗口对象的 addWindowListener 方法。从程序结构来看,这种处理方式最简单。

### 9.1.2 关于事件裁剪器

在此有必要对窗口事件裁剪器作出说明。窗口事件裁剪器是为方便程序员而专门设

计的类。定义如下：

```
public abstract class WindowAdapter extends Object implements WindowListener {  
    public WindowAdapter() // 构造方法  
    public void windowOpened(WindowEvent e) // 窗口被打开后调用  
    public void windowClosing(WindowEvent e) // 窗口被关闭时调用  
    public void windowClosed(WindowEvent e) // 窗口被关闭后调用  
    public void windowIconified(WindowEvent e) // 窗口被缩小为图标时调用  
    public void windowDeiconified(WindowEvent e) // 窗口被复原时调用  
    public void windowActivated(WindowEvent e) // 窗口激活时调用  
    public void windowDeactivated(WindowEvent e) // 窗口失活时调用  
}
```

从定义中可以看出,窗口事件裁剪器是一个抽象类,可接收窗口事件,其中的方法仅声明了返回值类型和参数类型。从这个类派生出来的子类可作为事件监听者,只需要为使用到的事件处理方法添加代码就可以了。由子类生成的对象具有事件监听功能,通过窗口的注册方法 `addWindowListener` 将这个对象注册给一个窗口,每当窗口事件发生时,就会被传递给监听对象来处理。

另一个常用的事件裁剪器是鼠标事件裁剪器,在 Java 中还有很多其他裁剪器,它们的原理都是相同的,用法也一样。这些事件裁剪器的出现,使程序员不用再为冗长的代码而发愁,编写程序更加简单,结构更加合理。

### 9.1.3 在窗口中加入组件

上面创建的窗口是空白的,你可在里面添加任意组件。通常的做法是在窗口里添加一个退出按钮,使窗口具有多种关闭功能。下面是一个例子。

**例 9.4** 在窗口中添加按钮,点击“显示”按钮可以显示文字,点击“退出”按钮可关闭窗口,同时窗口右上角的关闭按钮也起作用,如图 9.2。

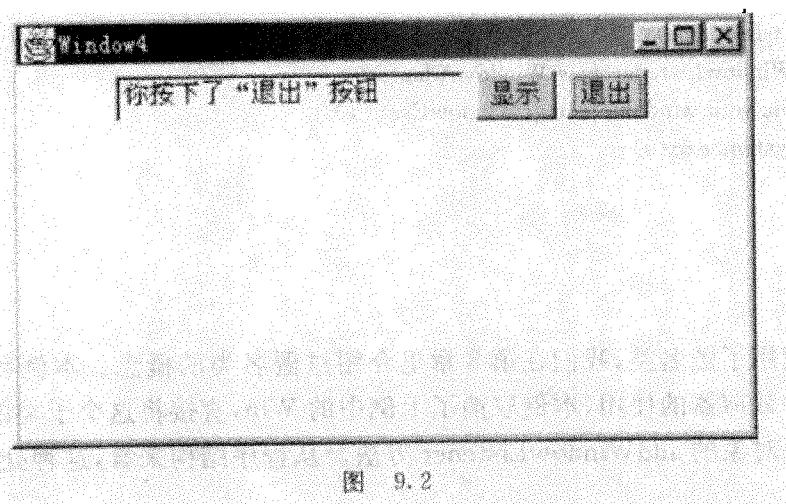


图 9.2

```
import java.awt.*;  
import java.awt.event.*;
```



```

public class W4 extends Frame implements ActionListener {
    Button btn1, btn2;
    TextField f;

    W4() {
        super("Window4");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setSize(350,200);
        btn1=new Button("显示");
        btn2=new Button("退出");
        f=new TextField(20);
        setLayout(new FlowLayout());
        add(f);
        add(btn1);
        add(btn2);
        btn1.addActionListener(this);
        btn2.addActionListener(this);
        show();
    }

    public static void main(String args[]) {
        new W4();
    }

    public void actionPerformed(ActionEvent e) {
        f.setText("你按下了" + e.getActionCommand() + "按钮");
        if (e.getSource()==btn2) {
            for (int i=0; i<100000000; i++);
            System.exit(0);
        }
    }
}

```

注意:Frame 类是容器,因此可以添加组件。show 方法或 setVisible 方法必须位于添加组件语句之后,否则组件显示不出来。

在程序中,为窗口本身注册了窗口事件监听者即匿名对象,为两个按钮注册的动作事件监听者为窗口本身。点击按钮时,首先在文本框里显示了事件源,然后判断事件源如果是 btn2 则调用系统的 exit 方法退出。我们在这里添加了一个循环进行延时,以便你可以看清点击“退出”按钮的显示结果。exit 方法的参数为 0 则可以正常退出,非 0 则为不正常退出。

#### 9.1.4 多重窗口

例 9.5 设计一个多重窗口的程序,如图 9.3。

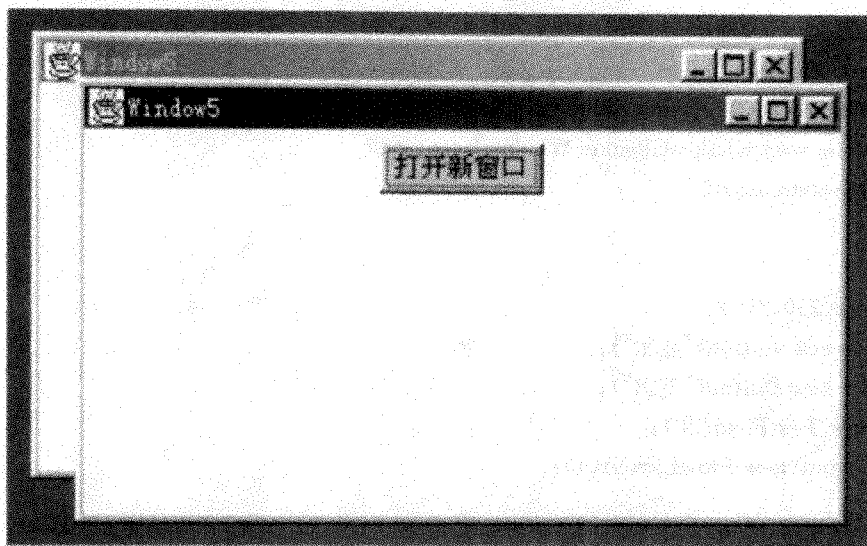


图 9.3

```
import java.awt.* ;
import java.awt.event.* ;
public class W5 extends Frame implements ActionListener {
    Button btn;
    W5() {
        super("Window5");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        setSize(350,200);
        btn=new Button("打开新窗口");
        setLayout(new FlowLayout());
        add(btn);
        btn.addActionListener(this);
        setLocation(200, 100);
        show();
    }

    public static void main(String args[]) {
        new W5();
    }

    public void actionPerformed(ActionEvent e) {
```

```

        new W5();
        setLocation(getX()+20, getY()+20);
        toFront();
    }
}

```

多重窗口设计的关键是哪个组件的动作能显示出新窗口,本程序中,按钮“打开新窗口”就具备这个功能。点击按钮后,将重复生成 W5 的新对象即窗口。如果想生成其他窗口,可在程序中另外定义一个类来生成不同的窗口。此外在下面介绍的菜单中,你也可以用一个菜单项来生成多重窗口。

## 9.2 菜单

菜单是图形用户界面的重要组成部分,由菜单条(MenuBar)、菜单(Menu)、菜单项(MenuItem)和复选菜单项(CheckboxMenuItem)等对象组成。

### 9.2.1 为窗口加入菜单

例 9.6 在窗口中添加菜单,如图 9.4。

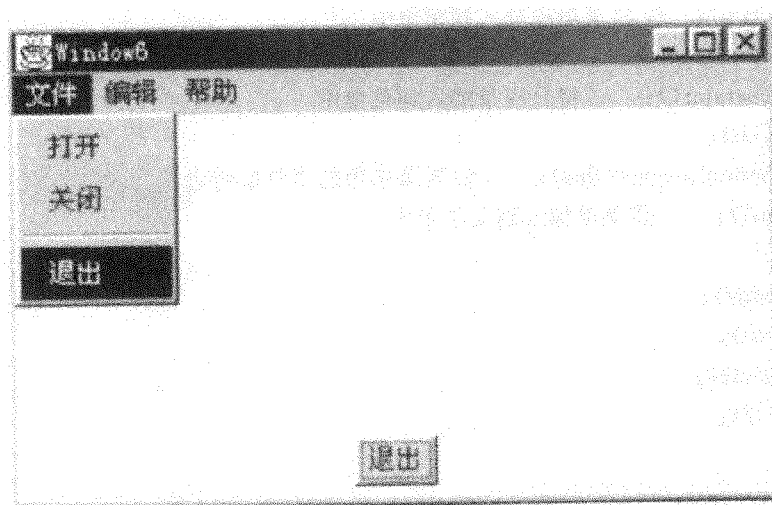


图 9.4

```

import java.awt.*;
import java.awt.event.*;
public class W6 extends Frame implements ActionListener {
    Panel p=new Panel();
    Button btn=new Button("退出");

    MenuBar mb=new MenuBar(); // 以下生成菜单组件对象
    Menu m1=new Menu("文件");
    MenuItem open=new MenuItem("打开");

```

```

MenuItem close=new MenuItem("关闭");
MenuItem exit=new MenuItem("退出");

Menu m2=new Menu("编辑");
MenuItem copy=new MenuItem("复制");
MenuItem cut=new MenuItem("剪切");
MenuItem paste=new MenuItem("粘贴");

Menu m3=new Menu("帮助");
MenuItem content=new MenuItem("目录");
MenuItem index=new MenuItem("索引");
MenuItem about=new MenuItem("关于");

W6() {
    super("Window6");
    setSize(350,200);
    add("South",p);
    p.add(btn);
    btn.addActionListener(this);

    m1.add(open); // 将菜单项加入到菜单中
    m1.add(close);
    m1.addSeparator(); // 将分隔条加入到菜单中
    m1.add(exit);
    exit.addActionListener(this); // 注册菜单项的事件监听者
    mb.add(m1); // 将菜单加入到菜单条中

    m2.add(copy);
    m2.add(cut);
    m2.add(paste);
    mb.add(m2);

    m3.add(content);
    m3.add(index);
    m3.addSeparator();
    m3.add(about);
    mb.add(m3);

    setMenuBar(mb); // 显示菜单条
    show();
}

public static void main(String args[]) {
    new W6();
}

```

```

    }

    public void actionPerformed(ActionEvent e) {
        if (e.getActionCommand() == "退出")
            System.exit(0);
    }
}

```

菜单程序的设计很简单,首先生成菜单组件各个部分的对象,然后进行菜单组装。菜单组装是指把菜单加到菜单条上,把菜单项加到菜单中。除了文字菜单项外,还可以在菜单中加入分隔条。菜单项具有动作事件,可以响应键盘和鼠标点击事件。将动作事件监听者注册给菜单项,就可由监听者处理菜单事件。

### 9.2.2 菜单综合应用

如果在菜单中加入另外一个菜单的话,就会变成多级菜单。菜单项可以添加快捷键,以便快速打开。菜单项也可以设计成复选菜单项,即用 `CheckboxMenuItem` 生成菜单项对象。

**例 9.7** 菜单综合应用,如图 9.5。

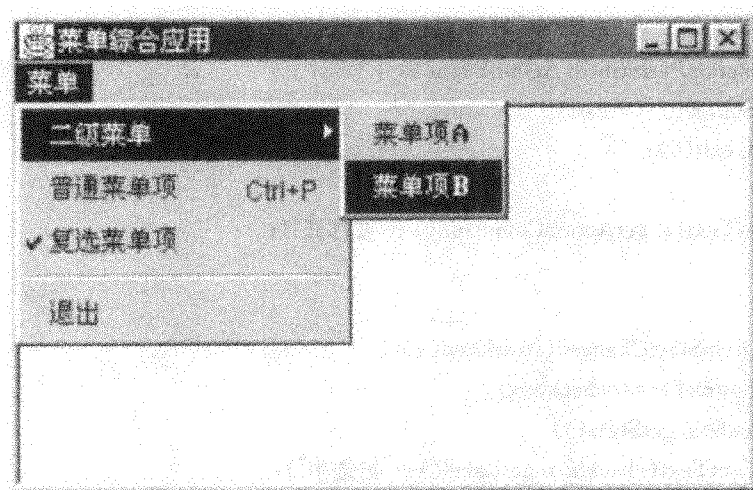


图 9.5

```

import java.awt.*;
import java.awt.event.*;

public class W7 extends Frame implements ActionListener, ItemListener{
    TextField msg=new TextField();
    MenuBar mb=new MenuBar();
    Menu m1=new Menu("菜单");
    Menu m2=new Menu("二级菜单");
    MenuItem item=new MenuItem("普通菜单项", new MenuShortcut('p'));
    CheckboxMenuItem checkbox=new CheckboxMenuItem("复选菜单项");
    MenuItem exit=new MenuItem("退出");
}

```

```

W7() {
    setTitle("菜单综合应用");
    setSize(350,200);
    add(msg);
    mb.add(m1);
    m1.add(m2); // 将二级菜单 m2 加入到 m1 中
    checkbox.setState(true); // 设定复选菜单项为选中
    m1.add(item);
    m1.add(checkbox);
    m1.addSeparator();
    m1.add(exit);
    m2.add("菜单项 A"); // 为二级菜单 m2 加入菜单项
    m2.add("菜单项 B");
    item.addActionListener(this); // 注册事件监听者
    checkbox.addItemListener(this);
    exit.addActionListener(this);
    setMenuBar(mb); // 显示菜单
    show(); // 显示窗口
}

public void actionPerformed(ActionEvent e) {
    if (e.getSource() == exit)
        System.exit(0);
    else
        msg.setText(e.getActionCommand()+"被打开");
}

public void itemStateChanged(ItemEvent e) {
    if (e.getSource() == checkbox)
        if (checkbox.getState())
            msg.setText(checkbox.getLabel()+"被选中");
        else
            msg.setText(checkbox.getLabel()+"被取消");
}

public static void main(String arg[]) {
    new W7();
}
}

```

说明：当用户按下快捷键 Ctrl+P 时，对应的菜单项“普通菜单项”就被打开。当用户点击复选菜单项时，它的前边就会出现“√”，表明已被选中，再点击则被取消。由于复选菜单项的功能区别于一般的菜单项，所以它提供了额外的方法，让用户来控制其状态。这

些方法是 `getState` 和 `setState`,前者取得该菜单项的状态,后者设置该菜单项的状态。

注意:普通菜单项相当于一个命令,而复选菜单项则不是命令。也就是说,它不产生动作事件,只产生状态改变事件。

### 9.2.3 弹出式菜单

弹出式菜单是一种非常方便的菜单工具,它平常依附在某个容器或组件上并不显现出来,当用户点击鼠标右键时它就会弹出来。

例 9.8 弹出式菜单应用,如图 9.6。

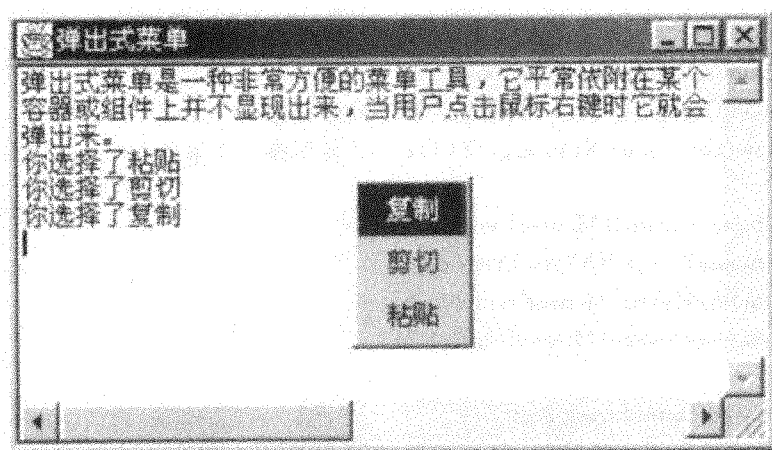


图 9.6

```
import java.awt.*;  
import java.awt.event.*;  
public class W8 extends Frame implements ActionListener, MouseListener {  
    TextArea msg=new TextArea();  
    PopupMenu pm=new PopupMenu();  
    MenuItem item1=new MenuItem("复制");  
    MenuItem item2=new MenuItem("剪切");  
    MenuItem item3=new MenuItem("粘贴");  
  
    W8() {  
        setTitle("弹出式菜单");  
        setSize(350,200);  
        addWindowListener(new WindowAdapter() { // 注册窗口的事件监听者  
            public void windowClosing(WindowEvent e) {  
                System.exit(0);  
            }  
        });  
        add(msg);  
        msg.add(pm); // 将弹出式菜单加入到文本区中  
        pm.add(item1);  
        pm.add(item2);  
    }  
}
```

```

        pm.add(item3);
        item1.addActionListener(this);
        item2.addActionListener(this);
        item3.addActionListener(this);
        msg.addMouseListener(this); // 注册文本区的鼠标事件监听者
        show();
    }

    public void actionPerformed(ActionEvent e) {
        msg.append("你选择了"+e.getActionCommand()+"\n");
    }

    public void mouseReleased(MouseEvent e) {
        if (e.isPopupTrigger()) // 判断是否按下鼠标右键
            pm.show(this, e.getX(), e.getY()); // 显示弹出式菜单
    }
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {}

    public static void main(String arg[]) {
        new W8();
    }
}

```

弹出式菜单由 `PopupMenu` 创建。程序中的弹出式菜单有 3 个菜单项,被加入到了文本区 `msg` 中。`W8` 实现了动作事件监听接口和鼠标事件监听接口,因为文本区能产生鼠标事件,菜单项能产生动作事件,这两个事件都要处理。

当用户在文本区点击鼠标左键或右键时,就触发了一个 `mousePressed` 事件。由于在这个事件的处理方法中,只有当点击了鼠标右键才能使 `isPopupTrigger` 方法取真值,所以点击鼠标左键不会弹出菜单。如果不用分支结构的话,无论点击鼠标左键或右键都会弹出菜单。`isPopupTrigger` 方法有 3 个参数,第一个是含有坐标系的对象,程序中用 `this` 来代表当前窗口对象,不能用 `msg`,因为文本区没有坐标系。后两个参数是鼠标点击的 `x` 和 `y` 位置,它们决定了弹出菜单的显示位置。

## 9.3 对话框

对话框是 GUI 中很常见的窗口对象,有着广泛的应用。对话框和普通窗口最大的不同就是对话框是依附在某个窗口上,一旦它所依附的窗口关闭了,对话框也要随着关闭。

### 9.3.1 自定义对话框

Java 提供了 `Dialog` 类用于制作自定义对话框,当你需要改变一些数据或需要一个提



示窗口时可使用自定义对话框。Dialog 有 7 个构造方法：

```
Dialog(Dialog owner)
Dialog(Dialog owner, String title)
Dialog(Dialog owner, String title, boolean modal)
Dialog(Frame owner)
Dialog(Frame owner, boolean modal)
Dialog(Frame owner, String title)
Dialog(Frame owner, String title, boolean modal)
```

其中,owner 是自定义对话框的所有者,它可以是一个 Dialog 或 Frame 对象。title 是对话框的标题,布尔变量 modal 取值为 true 则为模式对话框,否则为无模式(并存式)对话框。所谓模式对话框是指打开后必须作出响应的对话框,例如:

```
Dialog dlg=new Dialog(parent, "确认对话框", true);
```

这条语句可创建一个在 Windows 中常见的确认对话框,有“是”和“否”两个按钮,必须点击其中一个按钮,程序才能继续进行。

和 Frame 对象一样,对话框对象生成后需要调用 show 方法将其显示出来,hide 方法可隐藏对话框。常用的还有 getTitle 和 setTitle、isModal 和 setModal 等方法。

**例 9.9** 建立一个自定义对话框,如图 9.7 和图 9.8。

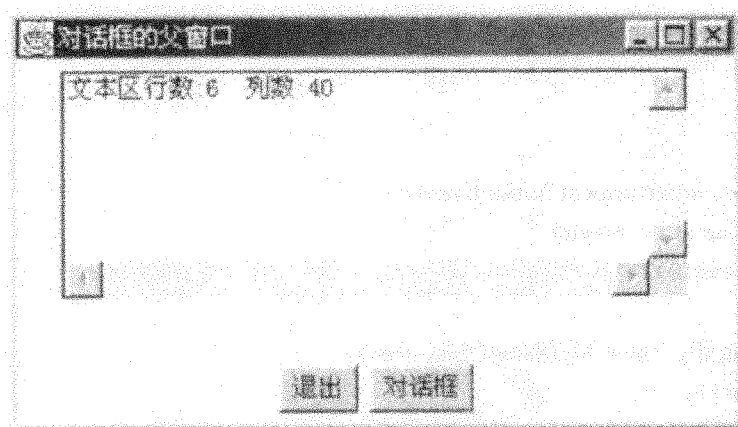


图 9.7

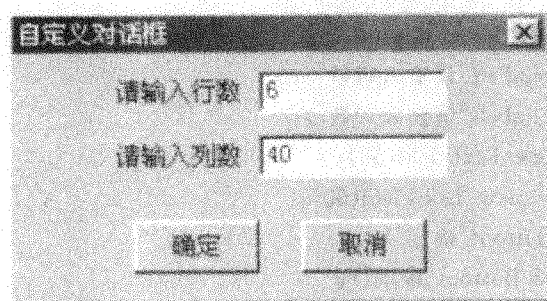


图 9.8

```

import java.awt.*;
import java.awt.event.*;
public class W9 extends Frame implements ActionListener {
    int row=6, col=40;
    Panel p1=new Panel(), p2=new Panel();
    TextArea ta=new TextArea("文本区行数:"+row+"列数:"+col, row, col);
    Button exit=new Button("退出"), dialog=new Button("对话框");

    W9() {
        setTitle("对话框的父窗口");
        setSize(350,200);
        add("Center", p1);
        add("South", p2);
        p1.add(ta);
        p2.add(exit);
        p2.add(dialog);
        exit.addActionListener(this);
        dialog.addActionListener(this);
        setVisible(true);
    }

    public static void main(String args[]) {
        new W9();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==exit)
            System.exit(0);
        else {
            MyDialog dlg=new MyDialog(this, true);
            dlg.show();
        }
    }

    class MyDialog extends Dialog implements ActionListener {
        Label label1=new Label("请输入行数");
        Label label2=new Label("请输入列数");
        TextField rows=new TextField(50);
        TextField columns=new TextField(50);
        Button OK=new Button("确定");
        Button Cancel=new Button("取消");

        MyDialog(W9 parent, boolean modal) {
            super(parent,modal);

```

```

setTitle("自定义对话框");
setSize(260,140);
setResizable(false);
setLayout(null);
add(label1);
add(label2);
label1.setBounds(50,30,65,20);
label2.setBounds(50,60,65,20);

add(rows);
add(columns);
rows.setText(Integer.toString(ta.getRows()));
columns.setText(Integer.toString(ta.getColumns()));
rows.setBounds(120,30,90,20);
columns.setBounds(120,60,90,20);

add(OK);
add(Cancel);
OK.setBounds(60,100,60,25);
Cancel.setBounds(140,100,60,25);
OK.addActionListener(this);
Cancel.addActionListener(this);
}

public void actionPerformed(ActionEvent e) {
    if(e.getSource() == OK) {
        int row=Integer.parseInt(rows.getText());
        int col=Integer.parseInt(columns.getText());
        ta.setRows(row);
        ta.setColumns(col);
        ta.setText("文本区行数:"+row+"列数:"+col);
    }
    dispose();
}
}
}

```

程序中的 W9 是主类, MyDialog 是 Dialog 的派生子类并作为 W9 的内部类, 内部类有一个好处是可以随意访问主类的成员。

点击主类中的“对话框”按钮可以打开自定义对话框。在 MyDialog 的构造方法中, 以主类为父窗口调用了父类的构造方法, 可生成一个有模式对话框。

此外, 程序中采用了手工布局方式来安排各组件的位置。两个文本框用来显示和输入数值, 点击“确定”按钮后, 用这两个数值设定主类中文本区的行列数。无论点击“确定”还是“取消”, 最后都调用 dispose 方法来关闭自定义对话框。

### 9.3.2 文件对话框

除了一般性对话框 `Dialog` 以外,还有一个常用的文件对话框类 `FileDialog`。当你需要打开或保存文件时,就可以使用文件对话框来输入文件名。使用不同的构造方法可以创建打开文件对话框或保存文件对话框:

```
FileDialog(Frame parent) // 生成一个打开文件对话框  
FileDialog(Frame parent, String title) // 生成带标题的打开文件对话框  
FileDialog(Frame parent, String title, int mode) // 生成带标题的打开或保存文件对话框
```

`parent` 和 `title` 与 `Dialog` 构造方法中的参数相同, `mode` 用于指定该文件对话框是打开文件还是保存文件,它只能从 `FileDialog` 类中定义的常量 `LOAD` 和 `SAVE` 中选取。

生成文件对话框后,需要调用 `show` 方法使其显示。然后,可以使用下面的一些方法进行文件操作。

```
public String getDirectory() // 从本文件对话框获取路径  
public void setDirectory(String dir) // 把指定目录设置为默认目录  
public String getFile() // 从本文件对话框获取文件  
public void setFile(String file) // 设置默认文件  
public FilenameFilter getFilenameFilter() // 返回当前文件过滤器  
public void setFilenameFilter(FilenameFilter filter) // 设置文件过滤器  
public int getMode() // 返回文件对话框的模式  
public int setMode() // 设置文件对话框的模式
```

下面的例子是一个简单的文本编辑器,利用文件对话框打开一个文件进行编辑,然后可以把它保存起来。

**例 9.10** 文件对话框的综合应用,图 9.9 为打开文件对话框、图 9.10 为文件内容、图 9.11 为保存文件对话框。

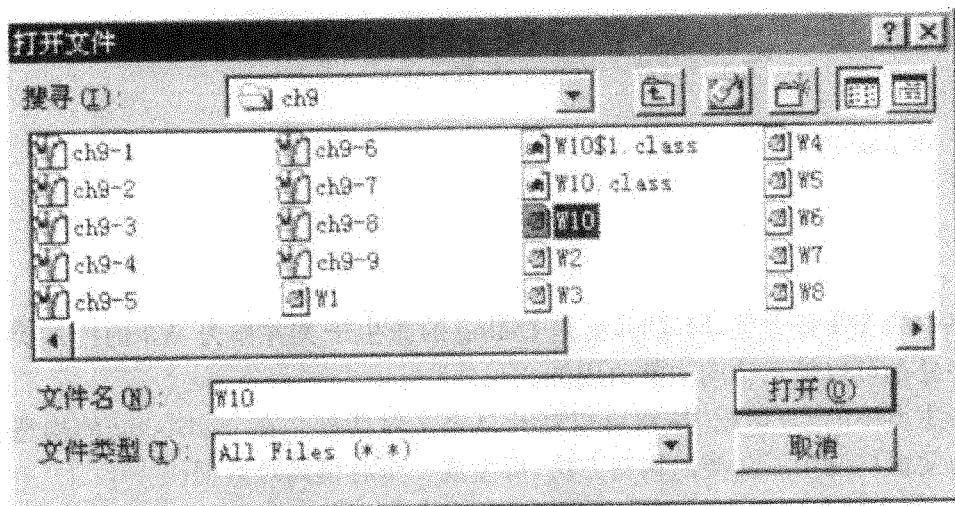


图 9.9

```

import java.io. * ;
import java.awt. * ;
import java.awt.event. * ;

public class W10 extends Frame implements ActionListener {
    FileDialog fileDlg;
    String str, fileName;
    byte byteBuf[]=new byte[10000];

    TextArea ta=new TextArea();
    MenuBar mb=new MenuBar();
    Menu m1=new Menu("文件");
    MenuItem open=new MenuItem("打开");
    MenuItem close=new MenuItem("关闭");
    MenuItem save=new MenuItem("保存");
    MenuItem exit=new MenuItem("退出");

    W10() {
        setTitle("简易文本编辑器");
        setSize(400,280);
        add("Center", ta);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        m1.add(open);
        m1.add(close);
        m1.add(save);
        m1.addSeparator();
        m1.add(exit);
        open.addActionListener(this);
        close.addActionListener(this);
        save.addActionListener(this);
        exit.addActionListener(this);

        mb.add(m1);
        setMenuBar(mb);
        show();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource()==exit)

```

```

        System.exit(0);
    else if (e.getSource() == close)    // 关闭文件
        ta.setText(null);              // 设置文本区为空
    else if (e.getSource() == open) {  // 打开文件
        fileDlg = new FileDialog(this, "打开文件"); // 生成文件对话框
        fileDlg.show();                  // 显示文件对话框
        fileName = fileDlg.getFile();    // 获取文件名
        try {
            FileInputStream in = new FileInputStream(fileName); // 建立文件输入流
            in.read(byteBuf);          // 将文件内容读到字节数组
            in.close();                 // 关闭文件输入流
            str = new String(byteBuf);  // 将字节数组转换成字符串
            ta.setText(str);            // 将字符串显示在文字区
            setTitle("简易文本编辑器 - " + fileName);
        } catch (IOException ioe) {}
    }
    else if (e.getSource() == save) { // 保存文件
        fileDlg = new FileDialog(this, "保存文件", FileDialog.SAVE); // 生成文件对话框
        fileDlg.show();
        fileName = fileDlg.getFile();
        str = ta.getText();          // 将文本区内容读至字符串
        byteBuf = str.getBytes();    // 将字符串转换成字节数组
        try {
            FileOutputStream out = new FileOutputStream(fileName); // 建立文件输出流
            out.write(byteBuf);    // 将字节数组写入文件输出流
            out.close();           // 关闭文件输出流
        } catch (IOException ioe) {}
    }
}

public static void main(String args[]) {
    new W10();
}
}

```

程序运行后,首先出现一个空白窗口。选取“文件”中的“打开”菜单项,将弹出文件对话框,如图 9.9 所示。从中选取要打开的文本文件如 W10.java,则文件的内容就会显示在文本编辑区,如图 9.10 所示。在文本区中,可以进行常规的编辑操作。尽管我们没有添加弹出式菜单,但你点击鼠标右键时还会有一个系统的弹出式编辑菜单出现。

选取“文件”中的“关闭”菜单项只是将文本区的内容清空,因为文件早已被输入流对象关闭了。

选取“文件”中的“保存”菜单项将打开保存文件对话框,如图 9.11 所示。可直接输入文件名,也可从文件显示区选择一个文件名,此时将弹出一个警告对话框,让你确认是否

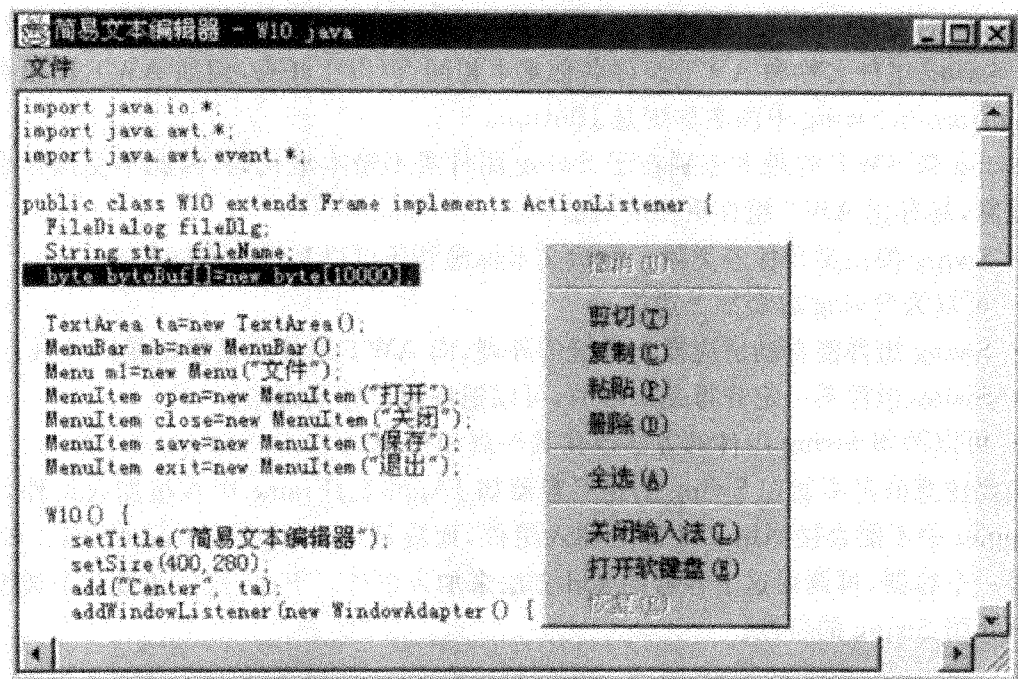


图 9.10

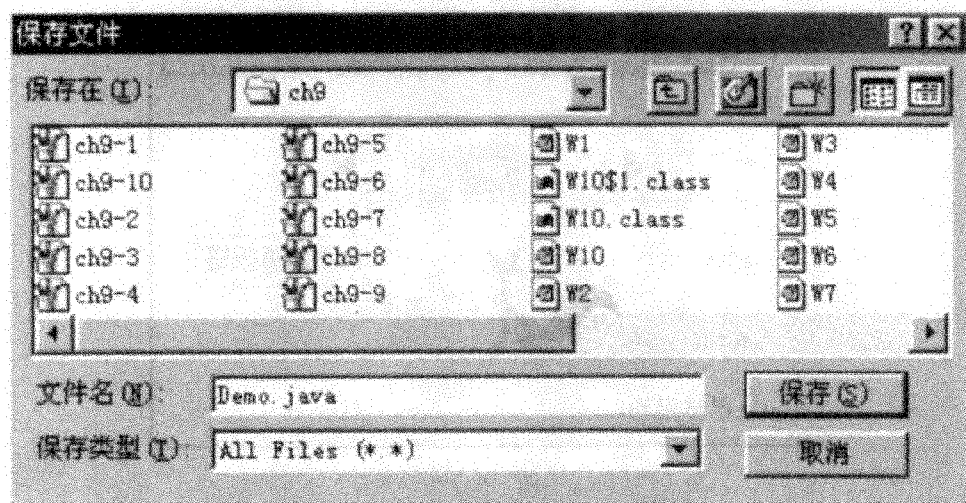


图 9.11

覆盖原文件。

本程序中使用了文件输入输出流,有关内容请参考后面的章节。

## 9.4 Swing 简介

在 Java 中,能够实现图形用户界面的类库有两个:java.awt 和 javax.swing。前者称为抽象窗口工具库 AWT(Abstract Windows Toolkit),从 JDK 1.1 开始提供,第 8 章和本章前面介绍的图形用户界面组件都是基于 AWT 的。后者是 Java 基础类库 JFC(Java

Foundation Classes)的一个组成部分,它提供了一套功能更强、数量更多的图形用户界面组件。Swing 组件名称和 AWT 组件名称基本相同,但以 J 开头,例如 AWT 按钮类的名称是 Button,在 Swing 中的名称则是 JButton。

Swing 和 AWT 的最大差别在于 Swing 组件类不带本地代码,因此不受操作系统平台的限制,具有比 AWT 组件更强的功能:

- Swing 按钮类和标签类除了显示文本标题外还可以显示图像标题;
- 可以为 Swing 容器加上边框;
- Swing 组件能自动适应操作系统的外观,而 AWT 组件总是保持相同外观;
- Swing 组件不一定非得是矩形的,可以把按钮设计成圆形;
- 可以调用 Swing 组件的方法改变其外观和行为。

需要注意的是不能在 Swing 的顶层容器如 JApplet、JFrame 中直接加入组件,例如,在 JApplet 中不能直接调用 add 方法加入组件,而应调用 JApplet 的 getContentPane 方法获得一个容器,再调用这个容器的 add 方法来加入组件。下面,我们通过几个典型的组件实例介绍 Swing 的特性。

#### 9.4.1 Swing 按钮与标签

例 9.11 图像按钮与图像标签应用,如图 9.12。



图 9.12

```
import javax.swing. * ;
import java.awt. * ;

public class W11 extends JApplet {
    Container pane;
    JPanel panell,panel2;
    JButton button1,button2,button3;
    JLabel label;

    public void init() {
        pane=getContentPane();
```



```

panel1=new JPanel(new FlowLayout());
panel2=new JPanel(new FlowLayout());
button1=new JButton(new ImageIcon("left.gif"));
button2=new JButton(new ImageIcon("go.gif"));
button3=new JButton(new ImageIcon("right.gif"));
label=new JLabel("图像标签",new ImageIcon("fox.gif"),SwingConstants.CENTER);

pane.setBackground(new Color(255,255,204));
panel1.setBackground(new Color(255,255,204));
panel2.setBackground(new Color(255,255,204));

button1.setToolTipText("向上翻页按钮");
button2.setToolTipText("跳转按钮");
button3.setToolTipText("向下翻页按钮");

pane.add(panel1,BorderLayout.NORTH);
pane.add(panel2,BorderLayout.SOUTH);
panel1.add(button1);
panel1.add(button2);
panel1.add(button3);
panel2.add(label);
}
}

```

上例中使用了 Swing 的 4 个组件, JApplet、JPanel、JButton、JLabel, 另外还用到了容器类 Container。

JApplet 需要自建容器以便把组件添加到容器对象中, 默认布局为边界布局。程序首先定义了容器对象 pane, 并在 init 方法中调用 getContentPane 方法创建了这个容器对象, 这样才能将其他组件加入到 JApplet 中。然后创建了 2 个面板对象并指定为流动布局, 创建了 3 个按钮对象并通过 ImageIcon 生成按钮图标。最后创建了一个既有文字又有图标的标签对象, 指定标签位置在显示区的中心, 文字紧跟在图标后面。

Swing 按钮除了可以显示图像外, 还可以显示文字提示。程序调用了 setToolTipText 方法分别为每个按钮设定了文字提示, 这样, 当鼠标指向按钮时就会显示出文字提示。

在其他方面, 如设定颜色、设定布局方式、添加组件以及组件事件响应等, JApplet 和 Applet 没有大的差别。

#### 9.4.2 Swing 工具栏

Swing 中的工具栏是一个很有用的组件, 它可以显示文字或图像按钮, 把一些常用的操作命令提供给用户。下面的例子介绍了如何创建工具栏并加入图像按钮。

**例 9.12** 工具栏与文本区应用, 如图 9.13。

```

import javax.swing. * ;
import java.awt. * ;
import java.awt.event. * ;

public class W12 extends JFrame implements ActionListener {
    JButton button1, button2, button3;
    JToolBar toolBar;
    JTextArea textArea;
    JScrollPane scrollPane;
    JPanel panel;

    public W12() {
        super("工具栏按钮");
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        button1=new JButton(new ImageIcon("left.gif"));
        button2=new JButton(new ImageIcon("go.gif"));
        button3=new JButton(new ImageIcon("right.gif"));
        button1.addActionListener(this);
        button2.addActionListener(this);
        button3.addActionListener(this);

        toolBar=new JToolBar();
        toolBar.add(button1);
        toolBar.add(button2);
        toolBar.add(button3);

        textArea=new JTextArea(6,30);
        scrollPane=new JScrollPane(textArea);

        panel=new JPanel();
        setContentPane(panel);
        panel.setLayout(new BorderLayout());
        panel.setPreferredSize(new Dimension(300,150));
        panel.add(toolBar,BorderLayout.NORTH);
        panel.add(scrollPane,BorderLayout.CENTER);

        pack();
        show();
    }
}

```

```

public void actionPerformed(ActionEvent e) {
    String s="";
    if (e.getSource()==button1)
        s="左按钮被单击\n";
    else if (e.getSource()==button2)
        s="中按钮被单击\n";
    else if (e.getSource()==button3)
        s="右按钮被单击\n";
    textArea.append(s);
}

public static void main(String[] args) {
    new W12();
}
}

```

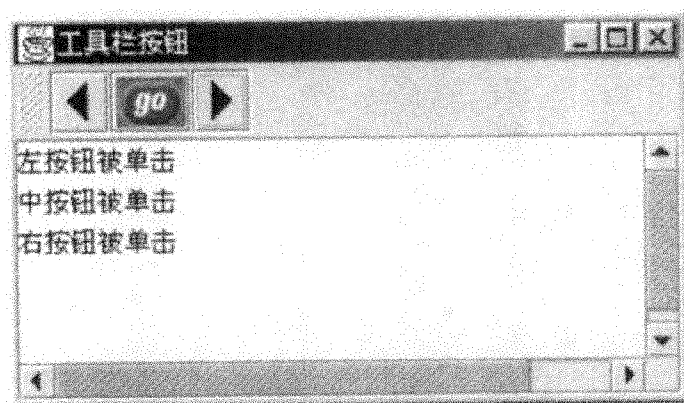


图 9.13

上例是一个 Java 应用程序,使用了 Swing 的 6 个组件:JFrame、JButton、JToolBar、JTextArea、JScrollPane、JPanel。

W12 的构造方法完成了大部分工作。首先创建了 3 个按钮,指定了相应的图像并注册了事件监听者。然后创建了一个工具栏并将 3 个按钮添加进去。Swing 中的文本区有所不同,它不能自动出现滚动框,因此常将它和 JScrollPane 连用,以 textArea 为参数创建滚动框对象。由图 9.13 可以看出,Swing 的滚动框更加精致。

和 JApplet 一样,JFrame 也是顶层容器,因此程序创建了一个 JPanel 对象作为基本容器来包容其他组件,并通过 setContentPane 方法将该对象指定为 JFrame 的基本容器。JPanel 的 setPreferredSize 方法可将面板区域设定为更合适的大小,使文本区自动出现滚动框。另外,pack 方法也是必不可少的,它使得窗口按照组件的大小来调整自身的大小。

actionPerformed 方法是对工具栏按钮点击事件的响应,点击一个按钮后,对应的字符串将显示在文本区中。

## 习 题

- 9-1 创建一个窗口对象时要注意什么? 创建一个可关闭的窗口。
- 9-2 创建一个通过按钮关闭的窗口。
- 9-3 向窗口添加菜单有哪几个步骤? 设计一个有“退出”命令的菜单并添加到窗口。
- 9-4 创建一个窗口,要求有“退出”按钮、菜单“退出”命令、窗口本身的关闭按钮也起作用。
- 9-5 如何为菜单添加分隔线? 如何创建复选菜单项? 如何创建多级菜单?
- 9-6 创建一个带有多级菜单和复选框菜单的窗口,并在菜单中加入分隔线。
- 9-7 创建一个窗口,单击“提示”按钮可出现一个写有“你好!”文字的对话框。
- 9-8 设计一个模拟文本编辑器,能在窗口中显示文本内容、编辑修改后可保存文件。
- 9-9 将第 9-7 题按 Swing 特性改写。

# 第10章

## 图形处理

Java 所有与图形有关的功能都包含在 AWT 包里。AWT (Abstract Windows Toolkit) 是抽象窗口工具包的缩写, 支持窗口界面的创建、简单图形的绘制、图形化文本输出和事件监听。用户可利用 AWT 提供的类和方法, 在窗口上绘制各种各样的图形和文本, 增加界面的美观。

### 10.1 基本图形

基本图形包括点、线、圆、矩形等, 是构成复杂图形的基础。绘制基本图形要使用 AWT 中的 Graphics 类, 它提供了各种基本图形的绘制方法。

#### 10.1.1 直线

**例 10.1** 在窗口上随机绘制 10 条直线, 如图 10.1 所示。

```
import java.applet. Applet;
import java.awt. Graphics;

public class DrawLines extends Applet {
    public void paint(Graphics g) {
        int i, x1, y1, x2, y2;
        for (i=1; i<=10; i++) {
            x1=(int) (Math.random()*10);
            y1=(int) (Math.random()*200);
            x2=(int) (Math.random()*380);
            y2=(int) (Math.random()*200);
            g.drawLine(x1,y1,x2,y2);
        }
    }
}
```

根据题目的要求, 我们使用一组随机数来指定直线两个端点的坐标, 并采用循环结构

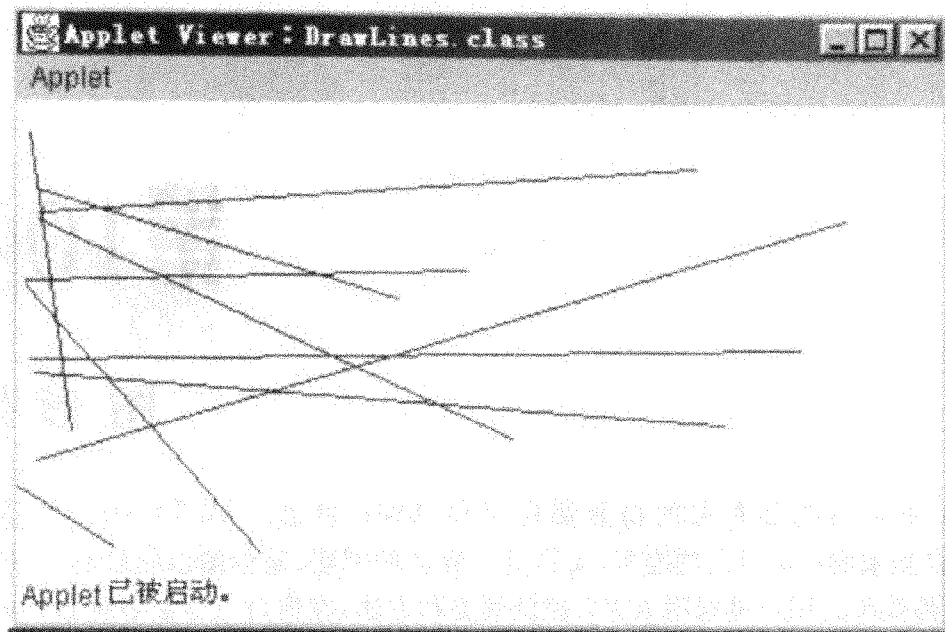


图 10.1

实现 10 次循环。

`drawLine` 方法中的 4 个整型参数代表直线两个端点的坐标。`random()` 是 `Math` 类中的一个方法, 该方法返回一个 `double` 值, 值域为  $[0.0, 1.0)$ 。`Math` 类是 `java.lang` 包的一部分, `java.lang` 可由编译器自动引入, 所以我们不必把 `Math` 类引入到程序中。直接调用 `random` 方法得到返回值太小, 在实际应用中往往乘以一个整数因子, 并强制类型转换为一个合适的整数。如 `(int)(Math.random() * 100)` 的实际取值范围为  $[0, 100]$ 。

在图形方式下要想准确定位, 必须了解屏幕坐标系的构成。Java 定义一个窗口工作区的左上角为坐标原点  $(0, 0)$ , 以像素点为单位, 顺序向右和向下延伸。图形的大小如超过窗口, 则超出部分不会显示出来。

### 10.1.2 矩形

**例 10.2** 利用 `Graphics` 的方法, 在窗口中画出若干个矩形, 结果如图 10.2 所示。

```
import java.applet. Applet;
import java.awt. Graphics;

public class Rectangles extends Applet {
    public void paint(Graphics g) {
        g.drawRect(20,30,80,80);
        g.fillRect(120,30,80,80);
        g.drawRoundRect(220,30,80,80,20,20);
        g.fillRoundRect(320,30,80,80,20,20);
    }
}
```

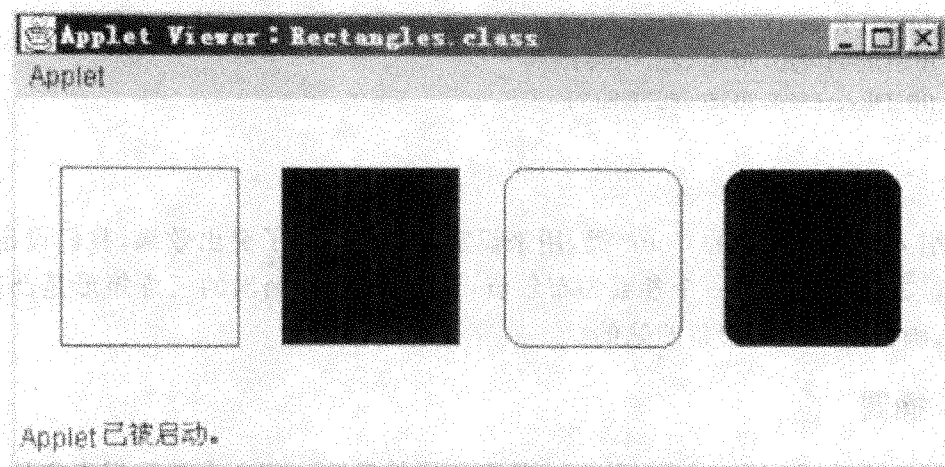


图 10.2

`drawRect` 方法可画一个矩形框,前 2 个参数指定矩形左上角的位置,后 2 个参数代表矩形的宽度和高度。`fillRect` 和 `drawRect` 的唯一不同之处在于,前者画出矩形框后用前景色填充。`drawRoundRect` 方法的前 4 个参数和 `drawRect` 的 4 个参数有相同的含义,后 2 个参数则代表了圆角的宽度和高度。如果想得到一个较为扁平的圆角,可取大一点的数值,反之,可取小一点的数值。`FillRoundRect` 则用填充前景色的方式画出圆角矩形。

例 10.3 利用 `Graphics` 的方法画三维矩形,结果如图 10.3 所示。

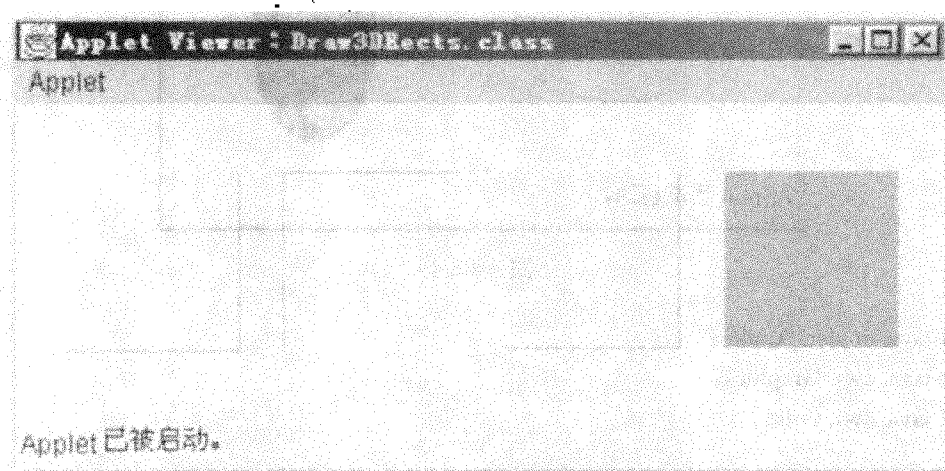


图 10.3

```
import java.applet. Applet;
import java.awt. Graphics;
import java.awt. Color;

public class Draw3DRects extends Applet {
    public void paint(Graphics g) {
        g. setColor(Color. yellow);
        g. draw3DRect(20,30,80,80,true);
    }
}
```

```

        g.draw3DRect(120,30,80,80,false);
        g.fill3DRect(220,30,80,80,true);
        g.fill3DRect(320,30,80,80,false);
    }
}

```

程序引入了 AWT 中的 Color 类,用来设定前景色。为了突出效果,我们设定当前颜色为黄色。方法中的最后一个参数为布尔值,设为真值时,画出的三维矩形是凸起的,设为假值时,画出的三维矩形是凹陷的。

### 10.1.3 椭圆

画椭圆的方法 drawOval 和 fillOval 具有相同的参数。前 2 个参数用来定位,实际指定的是包围椭圆的矩形的左上角位置,后 2 个参数指定了椭圆的宽度和高度,如果取相同值,则画出的是正圆。我们用一个例子来说明这两个方法的使用。

**例 10.4** 用 drawOval 和 fillOval 方法分别画椭圆,结果如图 10.4 所示。

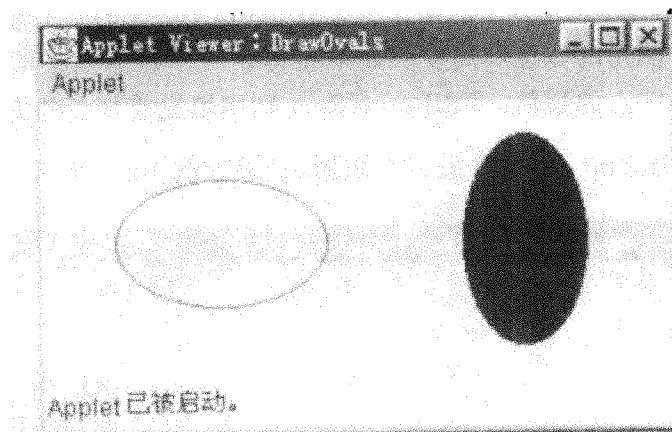


图 10.4

```

import java.applet. Applet;
import java.awt. Graphics;
import java.awt. Color;

public class DrawOvals extends Applet {
    public void paint(Graphics g) {
        g.setColor(Color. red);
        g.drawOval(35,35,100,60);
        g.setColor(Color. blue);
        g.fillOval(200,15,60,100);
    }
}

```

以上矩形和椭圆的画法有一定的规律性。画矩形框的方法 drawRect 是基本方法,其他方法中的前 4 个参数和 drawRect 的参数相同,都是决定矩形的位置和大小,那些不同



的参数则决定了所画图形的形状。

#### 10.1.4 圆弧

**例 10.5** 用 drawArc 和 fillArc 方法分别画圆弧,结果如图 10.5 所示。

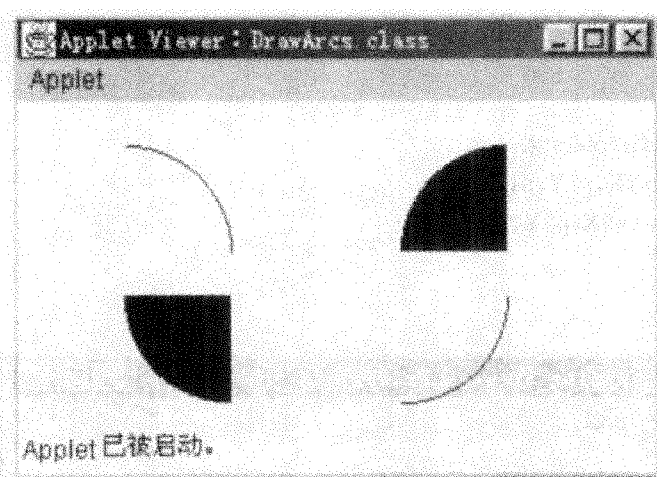


图 10.5

```
import java.applet.Applet;
import java.awt.Graphics;

public class DrawArcs extends Applet {
    public void paint(Graphics g) {
        g.drawArc(0,20,100,100,0,90);
        g.fillArc(180,20,100,100,90,90);
        g.fillArc(50,40,100,100,180,90);
        g.drawArc(130,40,100,100,0,-90);
    }
}
```

drawArc 方法可画圆弧,fillArc 方法实际画出的是扇形。圆弧是椭圆的一部分,夹在两个角之间,因此画圆弧的方法比画椭圆的方法多了两个参数:起始角和张角(以角度为单位)。起始角确定了圆弧的起始位置,张角确定了圆弧的大小,取正(负)值为沿逆(顺)时针方向画出圆弧。当张角取值大于 360 时,画出的就是椭圆。请读者注意程序中方法的后两个参数,起始角取不同的值,张角有正有负,分别画出了不同的圆弧和扇形。

#### 10.1.5 多边形

**例 10.6** 用 drawPolygon 和 fillPolygon 方法分别画多边形。结果如图 10.6 所示。

```
import java.applet.Applet;
import java.awt.Graphics;

public class DrawPols extends Applet {
```

```

int p1X[]={20,20,100,20};
int p1Y[]={20,80,20,20};
int p1=3;

int p2X[]={280,120,50,90,210,280};
int p2Y[]={20,50,100,110,70,20};
int p2=5;

public void paint(Graphics g) {
    g.fillPolygon(p1X,p1Y,p1);
    g.drawPolygon(p2X,p2Y,p2);
}
}

```

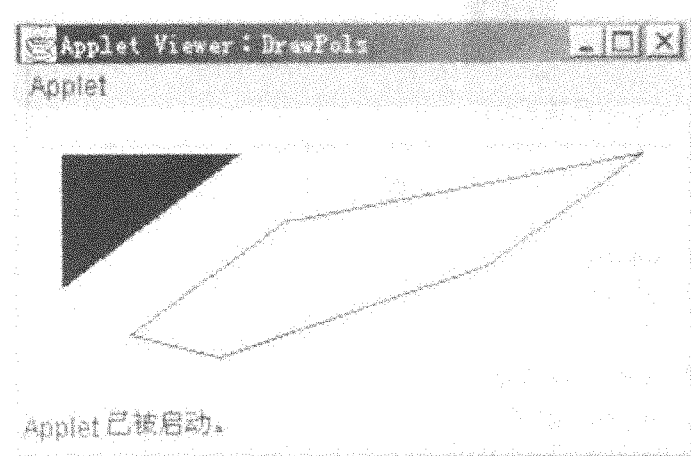


图 10.6 画多边形

多边形的多条边分别用两个整形数组表示  $x$  坐标和  $y$  坐标,并用一个整形数表示多边形的顶点数。多边形可以是封闭的也可以是开放的,取决于最后一条直线终点坐标的取值,若和第一条直线起点坐标重合,画出的多边形是封闭的,否则就是开放的。

## 10.2 画布

画布(Canvas)是用来绘图的组件,其作用和 Windows 的画图板类似,操作者可在画布上用鼠标直接画出图形。Canvas 继承自 Component 类,有自己的 Paint 方法,能响应鼠标和键盘事件。

为什么要使用画布呢?从上一节的内容里我们知道,通过调用 Graphics 的图形方法,可以画出各类图形。但这种直接画在窗口区的图形,很容易被其他组件覆盖掉。如果把图形画在画布上,就可解决覆盖问题。画布提供了一块专门的图形区域,通过设定自己的边界而和其他组件区分开,以保护画面不被覆盖。即使画面被破坏,也会自动恢复,即通过自己的 paint 方法重画出来。

### 10.2.1 创建画布

如何创建一个画布,并把它加入到程序中呢? 下面是一个 Applet 例子。

**例 10.7** 创建一个画布的简单程序,运行结果如图 10.7 所示。

```
import java.applet.Applet;
import java.awt.*;

public class Canvas1 extends Applet {
    public void init() {
        MyCanvas1 c=new MyCanvas1();
        c.setBackground(Color.green);
        c.setSize(300,200);
        add(c);
    }
}

class MyCanvas1 extends Canvas {
    public void paint(Graphics g) {
        g.fillOval(40,20,80,80);
    }
}
```

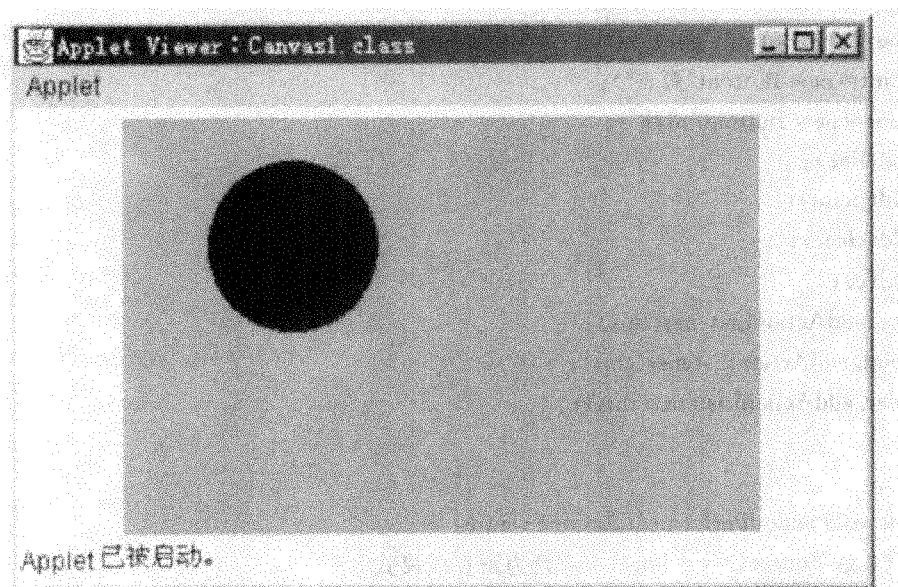


图 10.7

画布对象在创建时没有默认的大小,必须用 `setSize` 方法设定画布大小,否则在界面上将看不到画布。程序中的自定义类 `MyCanvas1` 是 `Canvas` 的子类,重载了 `Canvas` 的 `paint` 方法,该方法默认时没有任何功能。在画布上绘图,无论是用程序方法还是手工方法必须重载 `paint` 方法才能实现。

### 10.2.2 在画布上手工画图

手工画图时,一般使用鼠标或键盘。Canvas 通过监听鼠标或键盘事件,响应用户的画图操作。例 10.8 介绍了使用鼠标在画布上画直线和画点的编程方法,具有一定难度。用户点击“画线”按钮可画直线,点击“画点”按钮可画连续点,点击“清除”按钮可清除画面上的所有内容。本例中,先画一个由直线构成的旗子,再用画连续点的方法画出了几个字,结果如图 10.8 所示。

例 10.8 手工画图的复杂程序。

```
import java.applet. Applet;
import java.awt. * ;
import java.awt.event. * ;
import java.util. Vector;

public class Canvas2 extends Applet implements ActionListener {
    Button line, point, clear;
    MyCanvas2 c;

    public void init() {
        c=new MyCanvas2();
        c. setSize(350,200);
        c. setBackground(Color. green);
        line=new Button("画线");
        point=new Button("画点");
        clear=new Button("清除");
        add(line);
        add(point);
        add(clear);
        add(c);
        line. addActionListener(this);
        point. addActionListener(this);
        clear. addActionListener(this);
    }

    public void actionPerformed(ActionEvent e) {
        if (e. getSource()==line) // 设为画直线模式
            c. mode=0;
        else if (e. getSource()==point) // 设为画连续点模式
            c. mode=1;
        else if (e. getSource()==clear) { // 清除画面
            c. points=new Vector();
            c. x1= -1;
            c. repaint();
        }
    }
}
```



```

    }
}

class MyCanvas2 extends Canvas implements MouseListener, MouseMotionListener {
    int x1, y1, x2, y2, mode;
    Vector points=new Vector();

    MyCanvas2() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    public void paint(Graphics g) {
        for (int i=0; i<points.size(); i++) { // 所有操作结果被重新画出
            Rectangle r=(Rectangle)points.elementAt(i);
            g.drawLine(r.x, r.y, r.width, r.height);
        }
        if (x1!= -1 && mode ==0) //画当前直线
            g.drawLine(x1, y1, x2, y2);
    }

    public void mousePressed(MouseEvent e) { // 记录起点坐标
        x1=e.getX();
        y1=e.getY();
    }

    public void mouseDragged(MouseEvent e) {
        if (mode==0) { // 记录当前坐标
            x2=e.getX();
            y2=e.getY();
        }
        else { // 画连续点时保存每一个笔画的起点和当前坐标
            points.addElement(new Rectangle(x1, y1, e.getX(), e.getY()));
            x1=e.getX();
            y1=e.getY();
        }
        repaint();
    }

    public void mouseReleased(MouseEvent e) {
        if (mode==0) // 保存当前直线的起点和终点坐标
            points.addElement(new Rectangle(x1, y1, e.getX(), e.getY()));
    }
}

```

```

public void mouseClicked(MouseEvent e) {}
public void mouseEntered(MouseEvent e) {}
public void mouseExited(MouseEvent e) {}
public void mouseMoved(MouseEvent e) {}
}

```

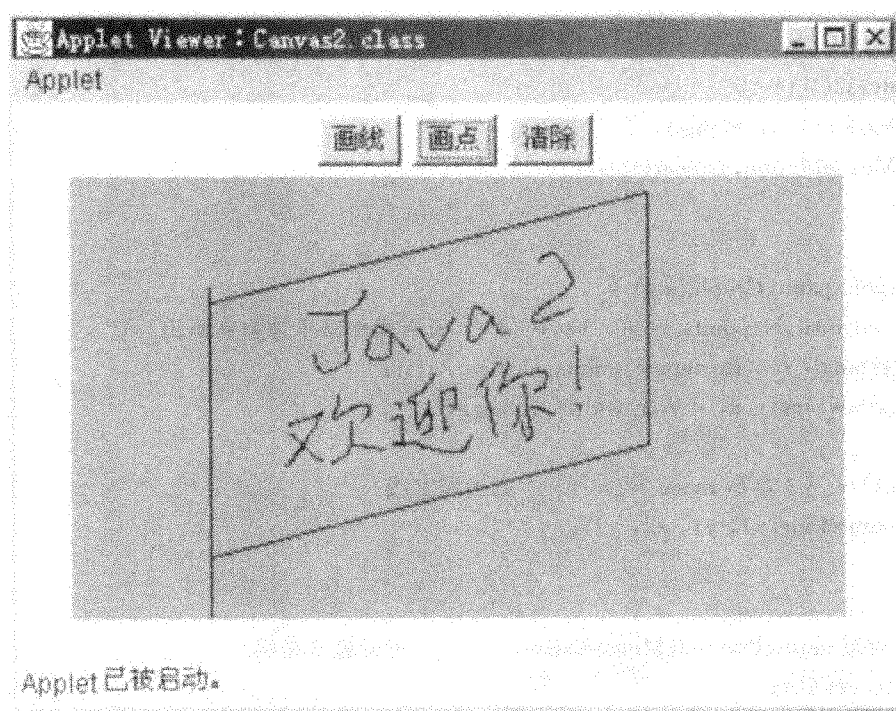


图 10.8

本例中,主类 Canvas2 负责添加三个按钮和一个画布,同时还负责监听按钮事件,根据用户点击的按钮设定画图模式或清画面。

设计了一个 MyCanvas2 类处理画图,负责监听鼠标事件,记录鼠标轨迹坐标,并把它们画出来。需要注意的是,随着鼠标的移动,轨迹坐标会越积越多,用什么变量把它们保存下来呢?这是 MyCanvas2 的一个核心问题,解决了这个问题,题目就迎刃而解了。因为只要给定类型和坐标,画图是很容易实现的。

MyCanvas2 中使用了向量而没有使用数组,因为数组是定长的,鼠标轨迹坐标的个数取决于用户,是一个未知数。向量类 Vector 可解决变长问题,向量相当于变长数组,但功能更强,可保存任意类型的数据包括对象。向量元素的个数取决于向其中添加元素的次数,而这种添加可以是任意多次的。

程序中声明了一个 Vector 实例 points,用来保存鼠标轨迹坐标。每当用户点击画面时,标志着一次画图的开始,MyCanvas2 就调用 mousePressed 方法记录鼠标当前位置,保存到 x1 和 y1 中。

当用户拖动鼠标时,调用 mouseDragged 方法,如果处于画线模式,鼠标当前位置就保存到 x2 和 y2 中,这就给画直线准备好了 4 个坐标。

如果处于画点模式,首先把起始位置 `x1`、`y1` 和当前位置 `e.getX()`、`e.getY()` 作为 `Rectangle` 的一组参数,整体添加到向量 `points` 中作为一个元素。然后把当前位置保存到 `x1` 和 `y1` 中作为下一次拖动的起点位置,最后调用 `repaint` 方法进行重画。

当用户松开鼠标时,意味着一次画图的结束,调用 `mouseReleased` 方法将画线模式下的起始位置 `x1`、`y1` 和当前位置 `e.getX()`、`e.getY()` 整体添加到向量 `points` 中,而画点模式下的起点和终点位置是随时保存的,此时就不再添加了。

我们再来看如何将保存的鼠标轨迹坐标画出来。在 `paint` 方法中,向量元素个数可由向量方法 `size` 得知,每个元素都是一个四边形,所以声明一个四边形 `r` 用来取出向量中的元素。之所以采用四边形作为保存鼠标轨迹坐标的工具,是因为四边形对象正好具有 4 个整型属性即起点位置 `x`、`y` 和宽(`width`)、高(`height`),可容纳一次画图的 4 个坐标值 `x1`、`y1`、`x2`、`y2`。利用循环把向量 `points` 保存的所有鼠标轨迹坐标都画出来,就可以看到多次画图的结果,否则,画面上只会出现当前画图的结果。

最后是对画线模式下当前直线的处理。画当前直线时,只有松开鼠标后,直线才能固定下来,并保存起点和终点坐标,在拖动过程中是不保存的,因此画当前直线时,直接使用记录了起始位置和当前位置的 4 个变量 `x1`、`y1`、`x2`、`y2`。

用户单击“清除”按钮时,向量 `points` 被重新初始化,并把 `x1` 赋值 -1,所以清除操作将使 `paint` 方法不产生任何画图动作,此时调用 `repaint` 方法的结果就是把画面清除干净。

## 10.3 文字输出

图形方式下的文字输出可实现多种效果。通过设定字体、风格和大小,实现文字输出的多样化。

### 10.3.1 字符串、字符和字节输出

在 `Graphics` 类中,Java 提供了三个文字输出方法,分别实现以字符串、字符和字节形式的文字输出。

- (1) 字符串输出方法 `drawString (String string, int x, int y)`
- (2) 字符输出方法 `drawChars (char chars[], int offset, int number, int x, int y)`
- (3) 字节输出方法 `drawBytes (byte bytes[], int offset, int number, int x, int y)`

例 10.9 文字输出方法示例,程序运行结果如图 10.9 所示。

```
import java.applet.Applet;
import java.awt.Graphics;

public class TextDemo extends Applet {
    String s="This is a string";
    char c[]={'这','是','一','个','字','符','数','组'};
    byte b[]={97,' ','b','y','t','e',' ',97,114,114,97,121};
```

```

public void paint(Graphics g) {
    g.drawString(s,30,30);
    g.drawChars(c,0,8,30,60);
    g.drawBytes(b,0,12,30,90);
}
}

```

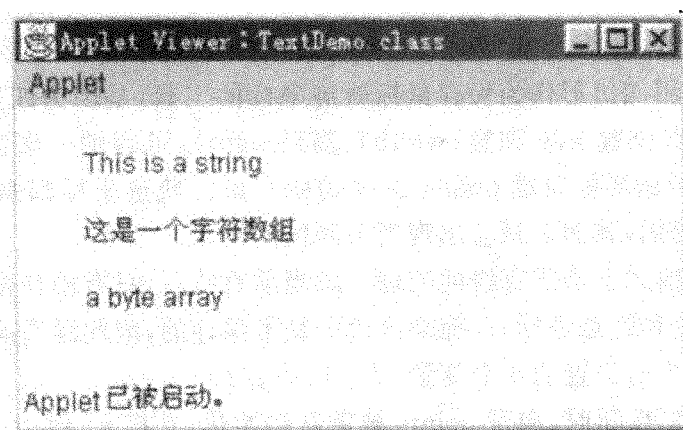


图 10.9

三种方法都是以当前字体和颜色在指定位置处输出文字。`drawString` 用于绘制一个字符串,使用时要传递给它三个参数:要输出的字符串 `s`, `x` 坐标和 `y` 坐标。字符串可以是常量也可以是变量。`x` 和 `y` 用于指定字符串左下角的位置,如果取 `(0,0)`,你将看不到这个字符串的输出。

`drawChars` 用于绘制字符型数组。当需要对一串字符采用不同字体或颜色绘制时,这个方法特别适合你。使用时,要分别指定字符数组、字符起始位置、要输出的字符个数、`x` 和 `y` 坐标。

`drawBytes` 用于绘制字节型数组,使用方法和 `drawChars` 类同。区别在于可显示字符的容量上,`drawChars` 中的字符数组是 16 位长的字符类型,可显示 Unicode 的全部字符;而 `drawBytes` 中的字节数组是 8 位长的有符号字节类型,只能显示 ASCII 字符集中的前 128 个字符。

### 10.3.2 字体控制

Java 在 `Font` 类中实现对字体的控制,可改变字体、风格和大小。字体的数量和计算机平台密切相关,不同计算机上安装的字体差别很大。

Java 提供了 5 种逻辑字体: `Dialog`、`SansSerif`、`Serif`、`Monospaced` 和 `DialogInput`,并将它们映射为计算机上的物理字体。如果使用了计算机不支持的字体,Java 将以该计算机的默认字体来代替。

**例 10.10** 显示字体风格和大小示例,程序运行结果如图 10.10 所示。

```
import java.applet.Applet;
```



```

import java.awt.*;

public class ShowFont extends Applet {
    Font font1=new Font("SansSerif",Font.BOLD,24);
    Font font2=new Font("Serif",Font.PLAIN,20);
    Font font3=new Font("Times New Roman",Font.PLAIN,20);

    public void paint(Graphics g) {
        g.setFont(font1);
        g.drawString("SansSerif 24 point BOLD",20,30);
        g.setFont(font2);
        g.drawString("Serif 20 point PLAIN",20,60);
        g.setFont(font3);
        g.drawString("Times New Roman 20 point PLAIN",20,90);
        g.setFont(new Font("Times New Roman",Font.ITALIC,20));
        g.drawString("Serif is equal to Times New Roman",20,120);
        g.setFont(new Font("宋体",Font.PLAIN,14));
        g.drawString("宋体 14 point PLAIN",20,140);
    }
}

```

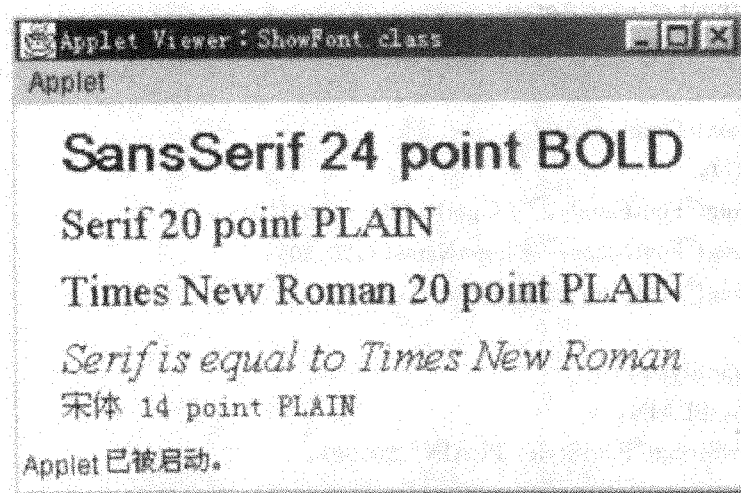


图 10.10

改变当前字体时,需要先创建一个 Font 对象,然后以该实例对象为参数调用 setFont 方法,这时的文字输出就以新字体显示。

Font 的构造方法包含三个参数:String fontName、int style、int size,分别表示字体名称、风格和大小。字体名称可使用逻辑字体名称,也可使用计算机现有的字体名称。例 10.10 分别使用了逻辑字体“SansSerif”、“Serif”和系统字体“Times New Roman”、“宋体”。其中,“Serif”映射了“Times New Roman”,二者是同一种字体。

字体有三种风格,BOLD(加重)、ITALIC(倾斜)和 PLAIN(正常)。字体大小可任选一个整数。有时,程序员需要获取当前字体的有关信息,如名称、风格和大小,例 10.11 给

出了一个应用范例。

例 10.11 获取字体信息示例,程序运行结果如图 10.11 所示。

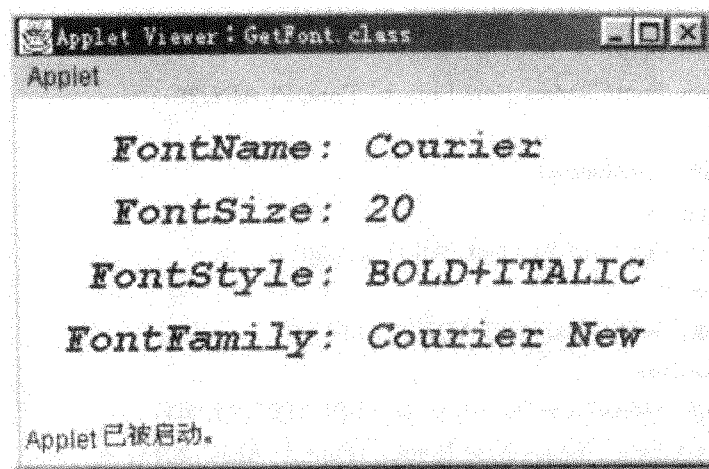


图 10.11

```
import java.applet.Applet;
import java.awt.*;

public class GetFont extends Applet {
    Font f=new Font("Courier",Font.BOLD+Font.ITALIC,20);

    public void paint(Graphics g) {
        g.setFont(f);
        g.drawString("FontFamily: "+f.getFamily(),20,120);
        g.drawString("FontName: "+f.getName(),20,30);
        g.drawString("FontSize: "+f.getSize(),20,60);

        switch (f.getStyle()) {
            case Font.PLAIN:
                g.drawString("FontStyle: PLAIN",20,90);
                break;
            case Font.BOLD:
                g.drawString("FontStyle: BOLD",20,90);
                break;
            case Font.ITALIC:
                g.drawString("FontStyle: ITALIC",20,90);
                break;
            default:
                g.drawString("FontStyle: BOLD+ITALIC",20,90);
        }
    }
}
```

字符类型的 `getFamily` 返回当前字体的家族名称;字符类型的 `getName` 返回当前字体名称;整数类型的 `getSize` 返回当前字体大小;整数类型的 `getStyle` 返回当前字体风格。

常量 `PLAIN`、`BOLD` 和 `ITALIC` 的值分别为 0、1 和 2,它们可以任意组合,但返回值为 3 时,一定是 `BOLD+ITALIC`。程序中采用多分支结构对 `getStyle` 的返回值进行判断,以正确显示字体风格。

## 10.4 颜色与绘图模式控制

图形方式下的文字输出和画图可实现多姿多彩的效果。通过设定颜色和绘图模式,可实现动态输出。

### 10.4.1 颜色控制

Java 在 `Color` 类中实现对颜色的控制,可改变前景色和背景色。Java 将常用颜色定义为颜色常量,如表 10.1 所示。

表 10.1 常用颜色表

颜色常量	色彩	RGB 值
black	黑色	(0, 0, 0)
blue	蓝色	(0, 0, 255)
cyan	青色	(0, 255, 255)
darkGray	深灰色	(64, 64, 64)
gray	灰色	(128, 128, 128)
green	绿色	(0, 255, 0)
lightGray	浅灰色	(192, 192, 192)
magenta	洋红色	(255, 0, 255)
orange	橙色	(255, 200, 0)
pink	粉红色	(255, 17, 175)
red	红色	(255, 0, 0)
white	白色	(255, 255, 255)
yellow	黄色	(255, 255, 0)

此外,程序员还可以通过调配三原色的比例创建自己的 `Color` 对象。`Color` 类有以下几种常用的构造方法:

- `Color(float r, float g, float b)` 指定三原色的浮点值,每个参数取值在 0.0~1.0 之间;
- `Color(int r, int g, int b)` 指定三原色的整数值,每个参数取值在 0~255 之间;

- `Color(int rgb)` 指定一个整型数代表三原色的混合值,16~23 比特位代表红色,8~15 比特位代表绿色,0~7 比特位代表蓝色。

例 10.12 颜色设置示例,程序运行结果如图 10.12 所示。

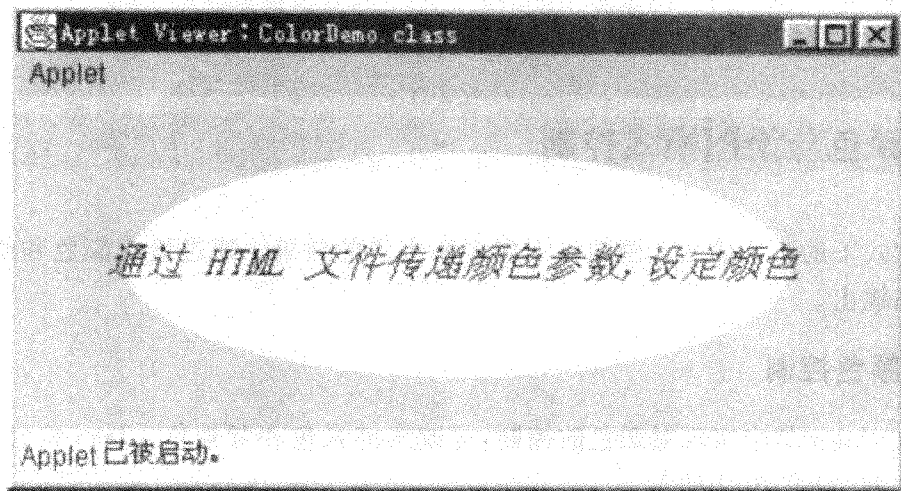


图 10.12

```
import java.applet. Applet;
import java.awt. * ;

public class ColorDemo extends Applet {
    int red, green, blue;
    String s;

    public void init() {
        s=getParameter("red");
        if (s!=null)
            red=Integer.parseInt(s);
        s=getParameter("green");
        if (s!=null)
            green=Integer.parseInt(s);
        s=getParameter("blue");
        if (s!=null)
            blue=Integer.parseInt(s);
        Color c=new Color(red, green, blue);
        setBackground(c);
        setForeground(Color. white);
    }

    public void paint(Graphics g) {
        g. fill(Oval(50,25,300,100);
        g. setColor(Color. blue);
    }
}
```

```

        g.setFont(new Font("宋体",Font.ITALIC,18));
        g.drawString("通过 HTML 文件传递颜色参数,设定颜色",40,80);
    }
}

```

ColorDemo 采取了特殊方法获取颜色值。在 init 方法中,通过 getParameter 方法读取 HTML 传递的参数,参数值即为设定的颜色值。这样做的好处是可在程序外面改变颜色而不必重新编译程序,修改 HTML 文件要比修改程序再重新编译简单得多,只需要在 HTML 文件中加入以下代码即可:

```

<PARAM name=red value=128>
<PARAM name=green value=255>
<PARAM name=blue value=128>

```

其中的 value 值可随时修改,相应地就改变了 Applet 的背景色。注意:HTML 的参数名和值都是字符串类型,要作相应的转换。

Color c=new Color(red, green, blue) 语句根据三原色的值合成颜色,创建实例对象 c 并用它设定 Applet 的背景色,前景色直接使用了颜色常量。在 paint 方法中,用前景色画出了一个白色的实心椭圆,用 setColor 方法重新设定前景色为蓝色,实际上是改变了画笔的颜色,最后输出的结果是一串蓝色的字符。

再看一个例子,输出一个字符串,其中每个字符都以不同的颜色显示。

**例 10.13** 多彩字符串,程序运行结果如图 10.13 所示。

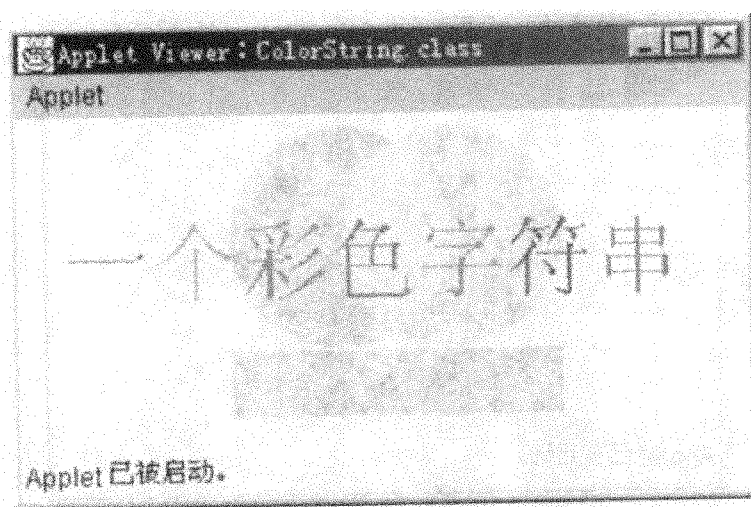


图 10.13

```

import java.applet. Applet;
import java.awt. * ;

public class ColorString extends Applet {
    char c[]={'一','个','彩','色','字','符','串'};
    int red,green,blue;

```

```

public void paint(Graphics g) {
    g.setFont(new Font("宋体",0,40));
    for (int i=0;i<7;i++) {
        red=(int)(Math.random()*255);
        green=(int)(Math.random()*255);
        blue=(int)(Math.random()*255);
        g.setColor(new Color(red,green,blue));
        g.drawChars(c,i,1,20+i*40,80);
    }
}
}

```

本例中,首先设定字体,然后用一个循环,每次生成三个随机数作为三原色的值,合成一个随机颜色,再逐个输出字符。读者可采用类似的方法,在程序中实现不同颜色、不同大小的字符串输出,使你的网页更加丰富多彩。

### 10.4.2 绘图模式控制

绘图模式实际上是指画笔的着色方式。默认的绘图模式为覆盖,后画的图形会覆盖已有的图形。Graphics 还提供了一个异或绘图模式,通过方法 `setXORMode` 可设定当前绘图模式为异或方式。下面通过一个例子看一下异或绘图模式有何不同。

**例 10.14** 异或绘图模式的使用,程序运行结果如图 10.14 所示。

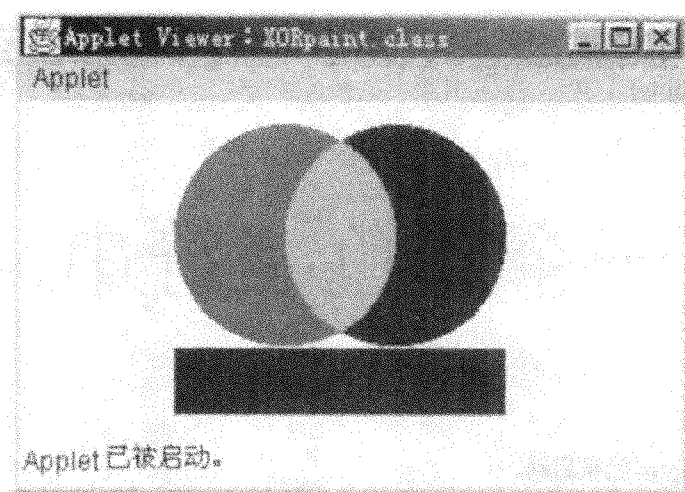


图 10.14

```

import java.applet.Applet;
import java.awt.*;

public class XORpaint extends Applet {
    public void paint(Graphics g) {
        setBackground(Color.white);
        g.setColor(Color.red);
    }
}

```

```

        g.fillOval(70,10,100,100);
        g.setXORMode(Color.green);
        g.fillOval(120,10,100,100);
        g.setPaintMode();
        g.setColor(Color.blue);
        g.fillRect(70,111,150,30);
    }
}

```

setXORMode 方法的参数是指定和当前色进行异或运算的颜色。绘图实际上是对视频内存赋值的过程,像素点的颜色由写入的数值决定。覆盖模式直接把颜色值写入视频内存,而异或模式则是把指定色和当前色进行异或运算,得到一个新颜色值,然后用新颜色和底色再进行异或运算,得到最后结果。例如红色(255,0,0)和绿色(0,255,0)在16~23位和8~15比特位上不同,在0~7比特位上相同,异或的结果是(255,255,0)即黄色。

程序中设定背景色为白色,前景色为红色并画出了第一个椭圆,然后设定异或颜色为绿色,这时画出的第二个椭圆却有两种颜色:绿色和蓝色。这是因为当前色和指定色进行按位异或运算(红色值-绿色值)后的结果为黄色,和两个椭圆交叉部分的底色进行按位异或运算(黄色值-红色值)后的结果为绿色,和其他部分的底色进行按位异或运算(黄色值-白色值)后的结果为蓝色。

恢复默认覆盖模式可通过调用 setPaintMode 方法,它是一个无参函数。下面的一个例子是一个简单的动画,利用异或绘图模式的原理,实现图形的移动。

**例 10.15** 利用异或绘图模式实现动画,程序运行结果如图 10.15 所示。

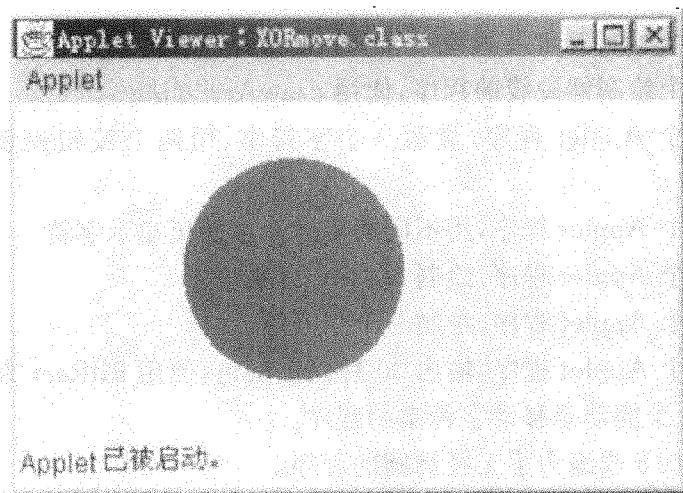


图 10.15

```

import java.applet.Applet;
import java.awt.*;

public class XORmove extends Applet {

```

```

public void paint(Graphics g) {
    setBackground(Color. white);
    setForeground(Color. red);
    g. fillOval(10,25,100,100);
    g. setXORMode(Color. white);
    for (int i=0;i<180;i++) {
        g. fillOval(10+i,25,100,100);
        g. fillOval(10+i+1,25,100,100);
    }
}
}

```

程序中,设定背景色为白色,前景色为红色,画出一个红色的圆。异或绘图模式的指定颜色为白色,和背景色相同,可起到擦除图形的作用。

在循环中,第一个圆是画在上次画出的圆的相同位置,异或绘图的结果是用白色覆盖了上次画出的圆,由于背景色也是白色,所以图形就消失了,起到了擦除的作用。第二个圆的位置右移了1个点,此时背景上已没有图形,画出的圆又成红色。每次循环重复相同的操作,图形就表现出动画的效果。

## 习 题

- 10-1 编写一个绘制5个同心圆的程序,分别使用 drawOval 和 drawArc 方法。
- 10-2 编写一个绘制螺旋线的程序,使用 drawArc 方法。
- 10-3 编写一个绘制五角星的程序,使用 drawPolygon 方法。
- 10-4 编写一个绘制螺旋线的程序,使用 drawArc 方法。
- 10-5 编写一个 Applet 程序,显示一个字符串,用两个按钮控制字符串的放大和缩小。
- 10-6 编写一个 Applet 程序,用不同的字体大小随机显示字符。
- 10-7 编写一个 Applet 程序,绘制 10×10 方格。
- 10-8 编写一个 Applet 程序,绘制一个立方体。
- 10-9 编写一个 Applet 程序,输出 30 种以上颜色,使用 fillRect 方法。
- 10-10 编写一个能动态移动字符串的程序。
- 10-11 将例 10.8 改编为手工画椭圆的程序。



# 第11章

## 多媒体编程

多媒体是信息世界的热点领域,而且有可能成为计算机最大的应用领域。多媒体的应用使程序更加丰富和生动,吸引人们走近计算机。图像、动画与声音处理是多媒体技术的主要内容,Java 语言提供了图像与声音类库,能够开发出功能强大的多媒体程序。本章主要介绍图像、动画与声音的编程。

### 11.1 图像处理

图像与上一章介绍的几何图形是有区别的。图形可以由程序画出来,而图像则是由专用软件生成的二进制文件,按不同格式存储图像数据就形成了不同的图像种类。Java 支持两种图像格式 JPEG 和 GIF。

#### 11.1.1 图像种类

图像具有独特魅力,能代替文字表达更丰富的内容,在程序中使用图像会达到很好的运行效果。图像一般用扫描仪配合图像处理软件制作成图像格式文件。图像的最终表达决定于屏幕像素点的颜色,即不同像素点取不同颜色就可以显示一幅图像。让哪一个像素点取何种颜色就是图像格式文件的任务。图像格式文件的扩展名代表着图像格式,也代表着图像还原处理方法。在进行图像处理之前,我们先来了解一点常用图像格式的基本知识。

##### 1. BMP

BMP 是 Windows 的标准位图文件格式,含有固定数量的像素点颜色,可用 Windows 的画图程序打开。这种图像在放大时,会出现锯齿边缘,变得很不清晰。图像文件没有被压缩过,规模较大,不适合在 Internet 上使用,Java 不能显示这种图像。

##### 2. JPEG 或 JPG

JPEG 称为联合图像专家组(joint photographic experts group),可用浏览器打开。JPEG(或 JPG)图像格式一般用来显示照片和具有连续色调的图像,它能保存图像所有颜

色信息。JPEG 是一种压缩的文件格式,在打开时自动解压缩。由于压缩后的文件规模较小,成为 Internet 上广泛使用的图像格式,Java 可以显示这种图像。

### 3. GIF

GIF 称为图像交换格式(graphic interchange format),可用浏览器打开。GIF 图像是一种压缩文件格式,由于它能最大限度地减少文件转换时间,所以在 HTML 文件中常用于显示插图或图标。GIF 格式能有效减少文件大小,有利于在 Internet 上使用,Java 支持这种图像格式。

#### 11.1.2 图像的显示

例 11.1 在 Applet 中显示一幅图像,如图 11.1 所示。

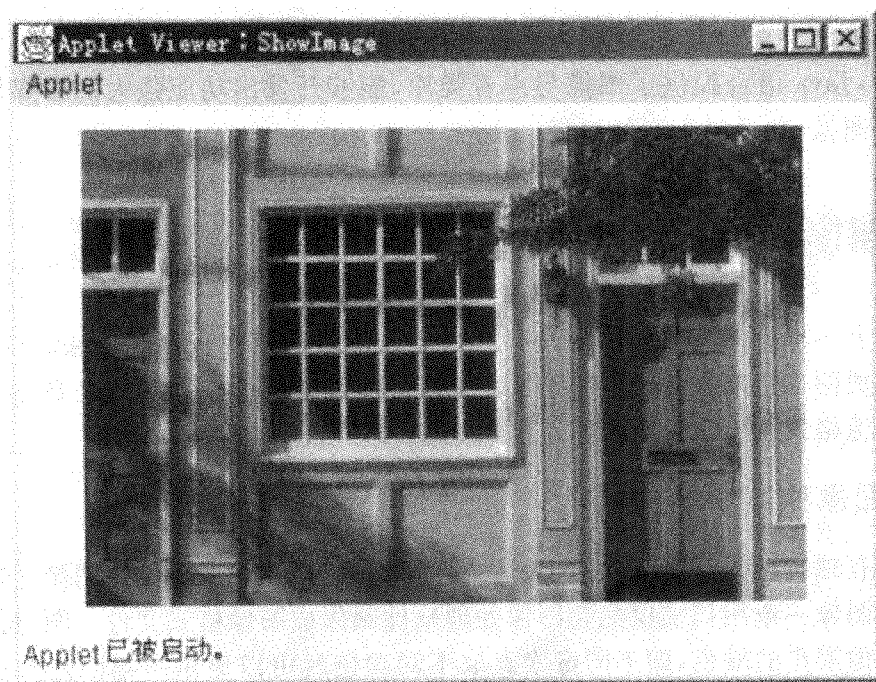


图 11.1

```
import java.applet.Applet;
import java.awt.Image;
import java.awt.Graphics;

public class ShowImage extends Applet {
    Image img;
    public void init(){
        img = getImage(getCodeBase(),"bld.jpg");
    }
    public void paint(Graphics g){
```

```

        g.drawImage(img,30,10,this);
    }
}

```

我们以这个程序为例,总结出在 Java 程序中显示一幅图像的步骤。图像显示分为两个步骤:首先加载图像(即将图像文件读入内存),然后画出图像。程序创建了一个 Image 对象 img,通过 Applet 的 getImage 方法加载图像文件 bld.jpg,将它和 img 联系起来。然后通过 Graphics 的 drawImage 方法显示图像。

## 1. 加载图像

加载图像一般放在初始化方法 init 中进行。程序中的 getImage 方法可加载 Java 支持的图像文件,它有两个参数,一个是图像文件地址,一个是图像文件名称。由于 Applet 是面向网络的,因此图像文件的存储位置并不局限于本地计算机的磁盘目录,大部分情况是直接读取 Web 服务器上的图像文件。getImage 方法返回一个 Image 对象,它的调用格式为:

```

Image getImage(URL url)
Image getImage(URL url, String name)

```

其中 url 是一个 URL 类的对象,代表一个网络地址(关于 URL 的概念请参考下一章的内容),例如下面的语句可以加载 sun 公司 Web 服务器指定位置上的一幅图片:

```

getImage(new URL("http://java.sun.com/graphics/people.gif"));

```

上述程序中没有直接给出一个具体网络地址,因为在不熟悉网站的情况下读取图像文件是毫无意义的。另一方面,设计 Applet 一般是为了显示自己已经准备好的图像,所以程序使用了 getCodeBase 方法。getCodeBase 返回的是 Applet 文件所在的地址,即该文件所在的目录,我们是把图像文件和 Applet 文件放在同一个目录下的,这样就能保证程序找到该图像文件。另外,使用 getDocumentBase 方法也可以达到同样目的。

## 2. 显示图像

显示图像需要调用 Graphics 类的方法 drawImage,它可以将 Image 对象关联的图像显示在 Applet 的指定位置。drawImage 方法的调用格式如下:

```

boolean drawImage(Image img, int x, int y, ImageObserver observer)
boolean drawImage(Image img, int x, int y, Color bgcolor, ImageObserver observer)

```

其中 img 就是要显示的图像、x 和 y 是图像显示位置(x 和 y 可取负值,表示一部分图像被移出了显示区)、bgcolor 是图像显示区域的背景色、observer 是图像加载跟踪器,通常将该参数指定为 this,即由 Applet 负责跟踪图像的加载情况。

这两种方法都是将图像照原样显示,能不能对图像进行缩放呢?使用下面两种调用格式就可以对图像进行缩放显示:

```

boolean drawImage(Image img, int x, int y, int width, int height, ImageObserver observer)

```

```
boolean drawImage(Image img, int x, int y, int w, int h, Color c, ImageObserver observer)
```

它们多出了两个参数 `width` 和 `height`, 即图像实际显示的宽度和高度。若原图像的宽度和高度与这两个参数值不一样时, Applet 会自动进行缩放, 使图像适合指定的矩形区域。

有时, 为了不使图像因缩放而变形失真, 可将原图的宽度和高度按相同的比例进行缩放。那么怎样知道原图的大小呢? 只需调用 `Image` 的两个方法就可以得到原图的宽度和高度。我们看下面的例子。

**例 11.2** 图像的缩放显示, 如图 11.2 所示。

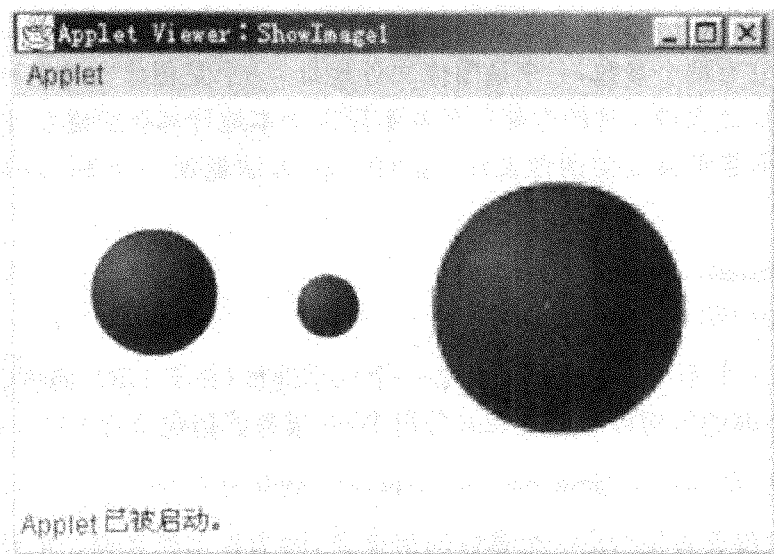


图 11.2 图像的缩放显示

```
import java.awt.* ;
import java.applet.* ;
public class ShowImage1 extends Applet {
    Image img;
    public void init(){
        img = getImage(getCodeBase(), "ball.jpg");
    }

    public void paint(Graphics g){
        int w = img.getWidth(this);
        int h = img.getHeight(this);
        g.drawImage(img, 20, 40, this);           // 原图
        g.drawImage(img, 120, 70, w/2, h/2, this); // 缩小一倍
        g.drawImage(img, 160, 0, w*2, h*2, this); // 放大一倍
    }
}
```

上述程序加载了一个圆球图像, 在 `paint` 方法中调用 `getWidth` 和 `getHeight` 方法取

得图像的宽度和高度。然后分别显示了原图、缩小一倍和放大一倍的图像。

### 11.1.3 幻灯机效果

如果 Applet 仅仅是显示一幅图像,没有什么特别的意义,不如直接在 HTML 文件中显示图像。Applet 应该做 HTML 做不到的事情,例如像幻灯机那样连续显示图像。

例 11.3 多幅图像的显示,如图 11.3 所示。

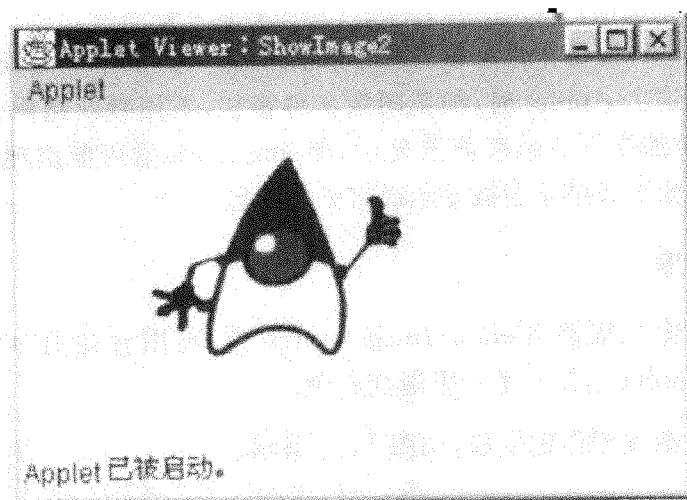


图 11.3

```
import java.awt.*;  
import java.awt.event.*;  
import java.applet.*;  
  
public class ShowImage2 extends Applet {  
    int index;  
    Image imgs[] = new Image[6];  
  
    public void init() {  
        addMouseListener(new MouseAdapter() {  
            public void mouseClicked(MouseEvent e) {  
                index = ++index % 6;  
                repaint();  
            }  
        });  
        for (int i = 0; i < 6; i++)  
            imgs[i] = getImage(getCodeBase(), "duke" + (i + 1) + ".gif");  
    }  
  
    public void paint(Graphics g) {  
        if (imgs[index] != null)  
            g.drawImage(imgs[index], 60, 20, this);  
    }  
}
```

在这个程序中,加载了 6 幅图像,点击鼠标可逐一显示图像,并在显示完 6 幅图像后自动返回第一幅重新开始。index 变量保存着当前显示的图像序号,imgs 是有 6 个元素的图像数组。在 init 方法中首先添加鼠标事件处理方法,通过鼠标事件裁剪器只实现了一个鼠标点击事件的处理方法,每次点击使 index 变量加 1 但不超过 5,并调用 repaint 方法重画图像。然后通过 for 循环合成文件名,分别加载 6 幅图像。

paint 方法很简单,就是在指定位置显示指定图像。方法中的判断结构另有用途,在本地计算机上运行这个 Applet 时,加载图像文件很快,不用判断结构也可以。但考虑到网络速度,图像有可能在下载结束前被显示,即 imgs 的元素可能出现 null 值,使用判断结构就可以避免图像下载结束前被显示而带来的问题。

#### 11.1.4 生成图像

Java 有一个图像生成器 MemoryImageSource 类,可用它在内存中生成一幅图像。下面的例子可在 Applet 上显示了一个渐变底色。

例 11.4 一个渐变图像的生成,如图 11.4 所示。

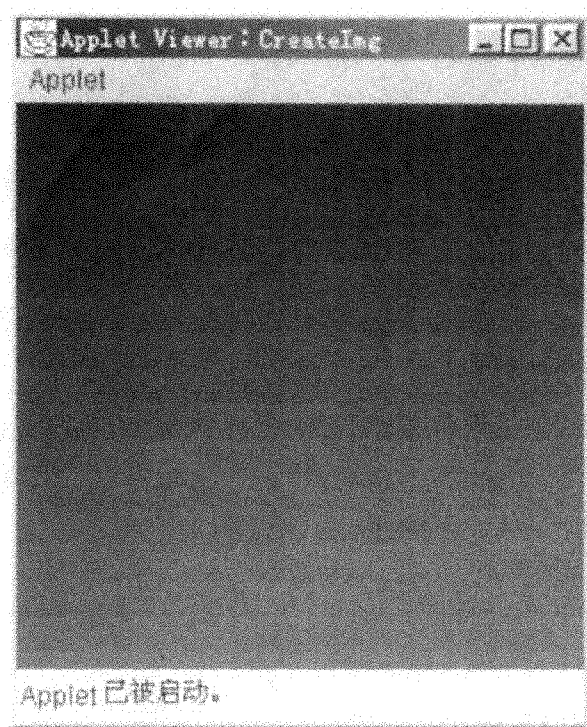


图 11.4 渐变图像

```
import java.awt.*;  
import java.awt.image.*;  
import java.applet.*;
```

```

public class show extends Applet {
    Image img;

    public void init() {
        int w=256; // 图像的宽度设为 256
        int h=256; // 图像的高度设为 256
        int[] pix=new int[w * h];
        int index=0;

        for (int red=0; red<h; red++) // red 从 0 变到 255
            for (int blue=0; blue<w; blue++) // blue 从 0 变到 255
                pix[index++]=(255<<24) | (red<<16) | blue;
        img=createImage(new MemoryImageSource(w, h, pix, 0, w));
    }

    public void paint(Graphics g) {
        g.drawImage(img,0,0,this);
    }
}

```

上述程序用整型数组 pix 生成一幅图像上每一点的颜色值。首先取得 Applet 的宽度和高度,二者的乘积代表了 Applet 区域中所有的点,将不同的点取不同的颜色,就可以得到一个渐变图像。外循环实现了 y 方向上由黑变红,内循环实现了 x 方向上由黑变蓝。表达式  $(255 \ll 24) | (red \ll 16) | blue$  的作用是将 255(0~7 位全为 1)左移至 24~32 位,将 red 值左移至 16~23 位,将 blue 值放入 0~7 位。每次循环,red 和 blue 都取不同的值,结果就产生了一个代表不同颜色值的整型数。

createImage 方法可以创建一个图像对象,它的参数是一个图像生成器,这个图像生成器由 MemoryImageSource 方法担任。该方法有 5 个参数:图像的宽度和高度、代表图像每一点颜色值的数组、画图像时的起始位置、扫描线的宽度。在 init 方法中,将这 5 个参数准备好后,生成了图像对象,最后在 paint 方法中将这个图像显示出来。

### 11.1.5 图形旋转与透明处理

在 Java AWT 中有一个增强图形类 Graphics2D,提供了对图形、图像和文本的特殊处理,可实现缩放、旋转、透明等效果。下面的例子演示了图形的旋转与透明处理。

**例 11.5** 图形的旋转与透明处理,如图 11.5 所示。

```

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.AlphaComposite;

public class Rotate extends java.applet.Applet {

```

```

public void paint(Graphics g) {
    g.setColor(Color. red);
    g.fillRect(100,30,100,100);
    Graphics2D g2=(Graphics2D) g; // 将 g 强制转换为 Graphics2D 类型
    int rule= AlphaComposite. SRC_ OVER; // 指定颜色合成模式
    float alpha=0.5f; // 指定颜色透明值
    AlphaComposite ac= AlphaComposite. getInstance(rule, alpha);
    g2.setComposite(ac); // 设定 g2 的颜色合成模式
    g2.setColor(Color. blue);
    g2.translate(150,10); // 转换 g2 的坐标系,平移到(150, 10)
    g2.rotate((45 * Math. PI)/180); // 绘图区顺时针旋转 45 度
    g2.fillRect(0,0,100,100);
}
}

```

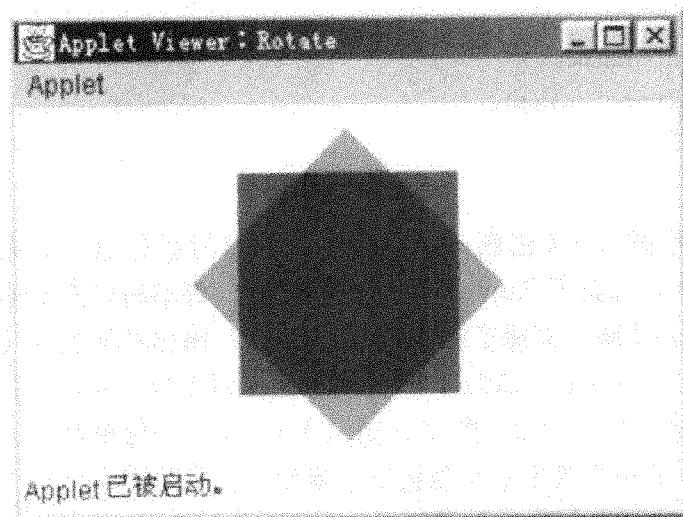


图 11.5

上述程序首先以正常方式画出一个红色矩形,然后以 2D 方式画出一个半透明并右旋 45 度的蓝色矩形。由于蓝色矩形是以 2D 方式画出的,所以两个图形叠放在一起,可以看到蓝色矩形下面的红色矩形。若以正常方式画出蓝色矩形,它将覆盖红色矩形。

以 2D 方式画图(包括图像),最重要的是将绘图区转换成 Graphics2D 类型,然后才能使用 Graphics2D 提供的各种方法。画透明图时,还要设定图形交叉区的颜色合成模式,这个模式由 AlphaComposite 类来生成,由 Graphics2D 来设定。

颜色合成模式指定为 SRC\_ OVER,即在目标图颜色基础上合成原图颜色,这个常量是 AlphaComposite 定义的,同时定义的还有多个常量。颜色的透明值取 0.5 即半透明,如果取 1 为完全不透明,如果取 0 则是全透明。

根据这两个参数值,调用 AlphaComposite 的 getInstance 方法创建模式对象 ac,然后根据 ac 调用 Graphics2D 的 setComposite 方法设定绘图区的颜色合成模式,此后将按照新模式画出图形。Graphics2D 的 rotate 方法以弧度为单位将绘图区顺时针旋转一个指定角度,由于旋转后坐标系发生变化,所以要先调用 translate 方法把坐标系平移,使画出



的图形保持在原坐标系的位置上。

## 11.2 动画处理

动画是指连续而平滑地显示多幅图像。动画的质量一方面取决于图像的质量,另一方面则取决于平滑程度。在计算机上,以 10~30 幅/每秒钟的速度显示动画即可达到满意的动画质量。在很多软件尤其是游戏软件的设计中,动画向程序员提出了挑战,但在 Java 中实现动画则是十分简单的事情。

下面让我们一起由浅入深地编写几个动画程序实例,并通过对这些实例的逐步改进来探讨 Java 动画技术的关键。

### 11.2.1 动画原理

其实,计算机动画原理十分简单,首先在屏幕上显示出第一帧画面,过一会儿把它擦掉,然后再显示下一帧画面,如此循环往复。由于人眼存在着一个视觉差,所以感觉好像画面中的物体在不断运动。

例 11.6 宇宙飞船游太空,如图 11.6 所示。

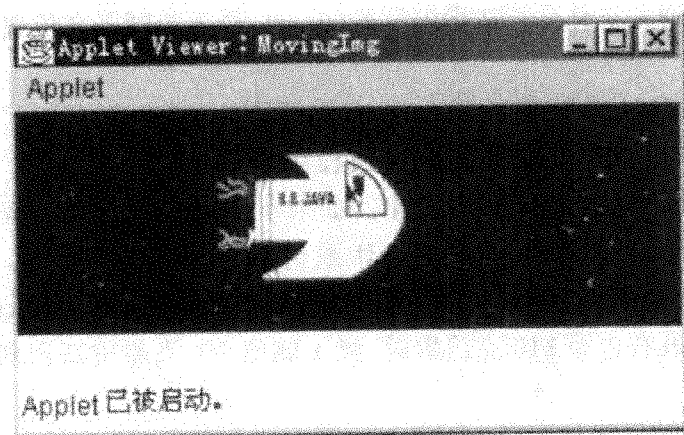


图 11.6

```
import java.awt.*;  
import java.applet.*;  
  
public class MovingImg extends Applet {  
    Image star, rocket;  
    int x=10;  
  
    public void init() {  
        star=getImage(getCodeBase(),"starfield.gif");  
        rocket=getImage(getCodeBase(),"rocket.gif");  
    }  
}
```

```

public void paint(Graphics g) {
    g.drawImage(star,0,0,this);
    g.drawImage(rocket,x,15,this);
    try {
        Thread.sleep(50);
        x+=5;
        if (x==210) {
            x=10;
            Thread.sleep(1000);
        }
    } catch (InterruptedException e) {}
    repaint();
}
}

```

这是一个很简单的动画,在 Applet 中有一个充当太空的背景图,一艘宇宙飞船在太空图上不断从左边移动到右边。

程序中创建了两个 Image 对象 star 和 rocket,在 init 方法中分别加载了两个图像文件和这两个对象关联起来。添加了变量 x 用来指定飞船的画出位置,x 初始化为 10。在 paint 方法中,注意到太空总是画在指定位置(0, 0),而飞船则画在位置(x, 15),其中 x 的值是不断变化的。

真正使飞船实现动画效果是在 try...catch 块中。程序调用了 sleep 方法,它是 Thread 类中定义的一个类方法,调用它可使程序休眠指定的毫秒数。sleep 方法会产生中断异常,因此必须放在 try...catch 块中。如果不调用 sleep 方法,程序将全速运行,必将导致换帧速度太快,画面闪烁严重。休眠时间设定为 50 毫秒,相当于换帧速度 20(1 000/50)。休眠结束后 x 的值加 5,意味着下一帧飞船画面的显示位置向右移动 5 个点。当飞船移动到最右边即 210 点位置时,将 x 赋值 10,飞船重新回到了左边,这是在 if 语句中实现的。

paint 方法的最后一条语句是调用 repaint 方法。repaint 方法的功能是重画图像,它先调用 update 方法将显示区清空,再调用 paint 方法画出图像。这就形成了一个循环,paint 调用了 repaint,而 repaint 又调用了 paint,使飞船不间断地来回移动。

运行这个 Applet 时,画面有闪烁现象。一般来说,画面越大,update 以背景色清除显示区所占用的时间就越长,不可避免地会产生闪烁。为了达到平滑而又没有闪烁的动画效果,就应该考虑采取一些补救措施。

覆盖 update 方法可以降低闪烁,但不能消除它。能有效消除闪烁的方法是采用图形双缓冲技术(graphics double buffering),请看下面改进后的程序。

### 11.2.2 图形双缓冲

例 11.7 改进后的宇宙飞船游太空,参见图 11.6。

```
import java.awt.*;
```

```

import java.applet. * ;

public class MovingImg1 extends Applet {
    Image star, rocket, buffer;
    Graphics gContext;
    int x=10;

    public void init() {
        star=getImage(getCodeBase(),"starfield.gif");
        rocket=getImage(getCodeBase(),"rocket.gif");
        buffer=createImage(getWidth(),getHeight());
        gContext=buffer.getGraphics();
    }

    public void paint(Graphics g) {
        gContext.drawImage(star,0,0,this);
        gContext.drawImage(rocket,x,15,this);
        g.drawImage(buffer,0,0,this);

        try {
            Thread.sleep(10);
            x+=2;
            if (x==210) {
                x=10;
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {}
        repaint();
    }

    public void update(Graphics g) {
        paint(g);
    }
}

```

改进后的程序比原程序增加了 buffer 和 gContext 对象,覆盖了 update 方法。换帧速度提高到 100(1 000/10),飞船每次移动 2 个点,动画效果更加平滑而且无闪烁。

buffer 是新增的 Image 对象,用作屏幕缓冲区。gContext 是新增的 Graphics 对象,代表着一个图形上下文即绘图区。在 init 方法中,程序调用 createImage 方法,按照 Applet 的宽度和高度创建了屏幕缓冲区,然后调用 getGraphics 方法创建了 buffer 的绘图区。

paint 方法改变了图像输出方向,两个图像都被画在屏幕缓冲区内。由于屏幕缓冲区不可见,使得画面交替时的闪烁现象也不可见。当屏幕缓冲区上的画图完成以后,再调用

drawImage 方法将整个屏幕缓冲区拷贝到屏幕上,这个过程是直接覆盖,不会产生闪烁。

图形双缓冲技术实际上是创建了一个不可见的后台屏幕,进行幕后操作,图像画在后台屏幕上,画好之后再拷贝到前台屏幕上。这种技术圆满解决了画面交替时的闪烁,但图像显示速度变慢,内存占用较大。

### 11.2.3 用线程实现动画

例 11.7 用图形双缓冲改善了图像闪烁问题,但仍存在一些其他问题。例如用户离开网页后,嵌入的 Applet 会继续运行,占用 CPU 时间。下面的例子出于网络实用的目的,采用独立线程实现动画。关于线程的内容请参考下一章。

例 11.8 用独立线程连续显示一个图像序列,如图 11.7 所示。

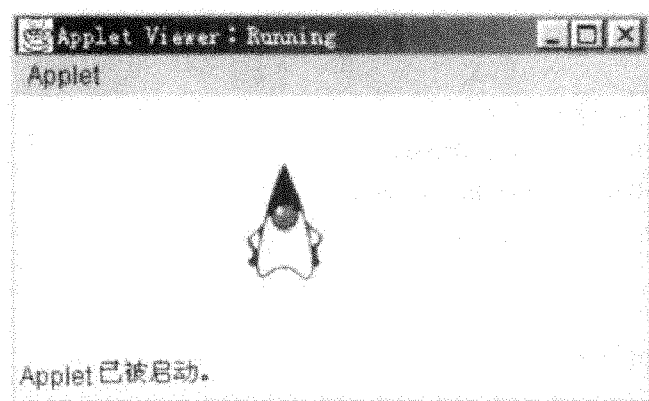


图 11.7

```
import java.awt. * ;
import java.applet. * ;

public class Running extends Applet implements Runnable {
    Image img[]=new Image[10];
    Image buffer;
    Graphics gContext;
    Thread animate;
    int index=0;

    public void init() {
        buffer=createImage(getWidth(),getHeight());
        gContext=buffer.getGraphics();
        for (int i=0;i<10;i++)
            img[i]=getImage(getCodeBase(),"T"+(i+1)+".gif");
    }

    public void start() {
        if (animate==null) {
            animate=new Thread(this);
```

```

        animate.start();
    }
}

public void stop() {
    if (animate != null) animate = null;
}

public void run() {
    while(true) {
        gContext.drawImage(img[index],100,20,this);
        repaint();
        try {
            animate.sleep(50);
        } catch (InterruptedException e) {}
        gContext.clearRect(100,20,100,100);
        index = ++index%10;
    }
}

public void paint(Graphics g) {
    g.drawImage(buffer,0,0,this);
}

public void update(Graphics g) {
    paint(g);
}
}

```

本程序加载了 10 个图像 (T1.gif ~ T10.gif)，采用了图形双缓冲技术，实现了 Runnable 接口中的 run 方法，这是一个和 Applet 同时运行的线程。对线程的控制由 Applet 的 start 和 stop 方法完成，Applet 运行时，就在 start 方法中启动线程，Applet 停止时，就在 stop 方法中停止线程。

对图像的操作全部放在 run 方法的永恒循环当中。首先调用 gContext 的 drawImage 方法把当前图像画在屏幕缓冲区内，怎样把它显示在屏幕上呢？是在 paint 方法中把屏幕缓冲区拷贝到屏幕上。但 paint 方法一般无法直接调用，因为要传递给它一个图形参数，所以通过调用 repaint 方法来间接调用 paint 以完成屏幕拷贝。repaint 方法无参数，它将调用 update 方法，由 update 方法调用 paint 方法并传递 g 参数。这就是我们曾介绍过的一个线程负责准备图像而另一个线程负责显示图像的动画方法。接下来，线程休眠 50 毫秒，然后清除屏幕缓冲区中的图像，将图像下标加 1 并取模。如果不清除屏幕缓冲区中的图像，将会出现图像重叠。下标加 1 后求余数，可保证取值范围总是 0 ~ 9。

#### 11.2.4 文字的动画显示

例 11.9 显示一个由小连续变大的字符串,如图 11.8 所示。

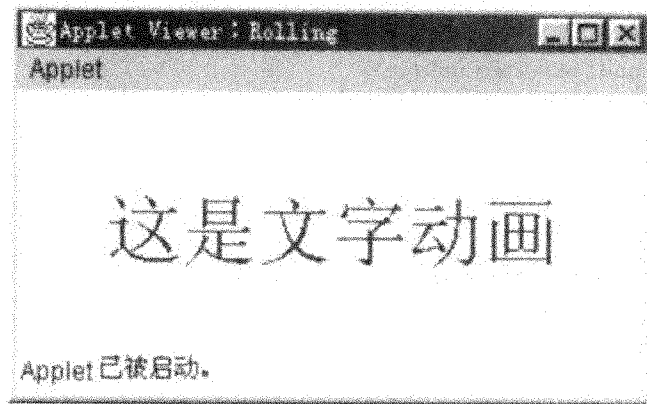


图 11.8

```
import java.awt.* ;
import java.applet.* ;

public class Rolling extends Applet implements Runnable {
    Image buffer;
    Graphics gContext;
    Thread animate;
    String s="这是文字动画";
    int w, h, x, y, size=12;

    public void init() {
        w=getWidth();
        h=getHeight();
        buffer=createImage(w,h);
        gContext=buffer.getGraphics();
        gContext.setColor(Color.blue);
    }

    public void start() {
        if (animate==null){
            animate=new Thread(this);
            animate.start();
        }
    }

    public void stop() {
        if (animate!=null)
            animate=null;
    }
}
```

```

    }

    public void run() {
        while(true) {
            x=(w-s.length()*size)/2;
            y=(h+size)/2;
            gContext.setFont(new Font("宋体",Font.PLAIN,size));
            gContext.drawString(s,x,y);
            repaint();
            try {
                animate.sleep(50);
            } catch (InterruptedException e) {}
            gContext.clearRect(0,0,w,h);
            if (++size>40)
                size=12;
        }
    }

    public void paint(Graphics g) {
        g.drawImage(buffer,0,0,this);
    }

    public void update(Graphics g) {
        paint(g);
    }
}

```

上述程序的设计思想和例 11.8 相同,只不过把图像改为字符串“这是文字动画”,通过设定字体大小和显示位置实现动画效果。

在 run 方法的永恒循环中,首先计算出字符串显示位置 x 和 y,使字符串每一次都显示在 Applet 的中心。调用 gContext 的 setFont 方法指定字体为宋体、字体风格为 PLAIN、字体大小为 size。调用 gContext 的 drawString 方法在指定位置输出字符串。然后调用 repaint 方法进行屏幕拷贝。线程休眠 50 毫秒后,清除后台屏幕中的图像。最后,对字体大小 size 进行处理,每次增量后,如果 size 大于 40 就恢复到初始值 12。

### 11.2.5 图像高级处理——水中倒影

**例 11.10** 用一幅图像制作出它的水中倒影,并能显示水波纹,如图 11.9 所示。

```

import java.awt.*;
import java.applet.*;

public class Lake extends Applet implements Runnable {
    Thread animate;
    Image img,buffer;

```

```

Graphics gContext;
int width,height;

public void init() {
    img=getImage(getCodeBase(),"tree.jpeg");
    MediaTracker tracker=new MediaTracker(this); // 创建图像加载跟踪器
    tracker.addImage(img,0); // 添加要跟踪的图像,代号为 0
    try {
        tracker.waitForID(0); // 等待图像加载完毕
    } catch (InterruptedException e) {}

    width=img.getWidth(this);
    height=img.getHeight(this)/2; // 仅使用图像的一半

    buffer=createImage(2 * width,height); // 创建后台屏幕,原始图像的两倍宽度
    gContext=buffer.getGraphics();
    gContext.drawImage(img,0,-height,this); // 图像的下半部分画到后台屏幕

    for (int i=0;i<height;i++) // 将图像逐线拷贝,生成图像倒影
        gContext.copyArea(0,i,width,1,width,(height-1)-2 * i); // 拷贝到后台屏幕右半边

    gContext.clearRect(0,0,width,height); // 清除后台屏幕左半边
}

public void start() {
    if (animate==null) {
        animate=new Thread(this);
        animate.start();
    }
}

public void stop() {
    if (animate!=null)
        animate=null;
}

public void run() {
    int dy,num=0;
    double d;

    while (true) {
        d=num * Math.PI/6; // 生成一个角度,共有 12 个值
        for (int i=0;i<height;i++) {
            dy=(int)((i/12.0D+1) * Math.sin(height/12.0D * (height-i)/(i+1)+d)); // 经验

```



公式

```
        gContext.copyArea(width,i+dy,width,1,-width,-dy); // 从右向左拷贝生成波纹
    }
    repaint();
    num=++num%12;
    try {
        Thread.sleep(50);
    } catch (InterruptedException e) {}
}

}

public void update(Graphics g) {
    paint(g);
}

public void paint(Graphics g) {
    g.drawImage(img,0,-height,this); // 显示图像的下半部分
    g.drawImage(buffer,0,height,this); // 显示图像倒影,合成一幅完整图像
}
}
```

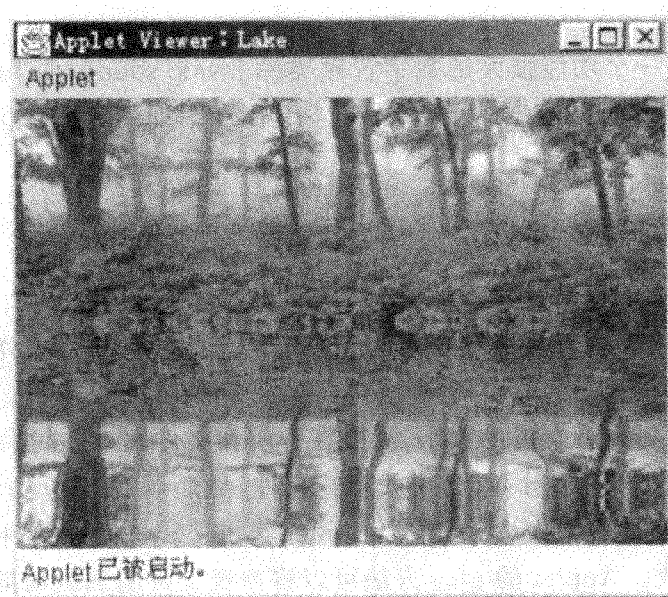


图 11.9

上述程序最关键的是如何生成图像的倒影以及如何水波纹显示倒影。设计思想是创建一个 2 倍宽度的后台屏幕,将原始图像的倒影放在右半部分,然后在线程中将图像倒影逐线拷贝到左边,并在拷贝过程中实现水波纹效果,最后将后台屏幕的左半部分即图像倒影和原始图像合成一幅完整图像显示出来。

首先创建一个图像加载跟踪器对象,使图像完全加载后再进行下一步的工作。调用跟踪器的 addImage 方法添加被跟踪的图像 img,然后调用跟踪器的 waitUntil 方法等待

img 完全被加载。取得图像宽度和高度后,创建一个 2 倍宽度和一半高度的后台屏幕 buffer,将原始图像的下半部分拷贝到 buffer 的左边,这是因为想减小图像的显示高度。

图像倒影的制作调用了 Graphics 的 copyArea 方法,将原始图像由上到下逐条线地拷贝到后台屏幕的右边,拷贝位置则是由下到上。copyArea 方法的前 4 个参数是要拷贝的图像位置和大小,后两个参数是相对于原位置的偏移量,取正值是从左向右从上到下拷贝,取负值则相反。

图像倒影的水波纹效果在线程 run 方法的永恒循环中实现。num 控制着 12 个角度的生成,代表着 12 个水波纹。从右向左逐线拷贝图像倒影时,调用了 Math 的正弦函数,根据经验公式生成一个 y 方向的偏移量 dy。这个偏移量有正有负,使拷贝到左边的每条线重新排列位置,构成了一个水波纹图像。角度的每次变化都生成一幅不同的倒影图像,由线程控制着每隔 50 毫秒显示一幅倒影图像,这就实现了水波纹效果。注意,线程操作的是后台屏幕,通过调用 repaint 方法完成从后台屏幕向前台屏幕的拷贝。

paint 方法具体实施写屏,首先将原始图像 img 的下半部分显示在屏幕上,图像显示位置 y 取 -height 值就可完成。然后将后台屏幕 buffer 的左半部分显示在 img 的下方,构成一幅完整的并且能水波纹显示的水中倒影。



JDK1.3 新增了一个功能强大的数字音频类库 javax. sound,使 Java 程序能录制、处理和播放声音和 MIDI 音频数据。限于篇幅,本节仅介绍如何利用 Applet 播放各种声音剪辑。

### 11.3.1 加载声音文件

Java 开发工具 JDK1.3 比以前版本支持更多的声音格式,共有 5 种: AIFF、AU、WAV、MIDI、RMF。音质可为 8 位或 16 位的单声道和立体声,采样频率从 8 kHz 到 48 kHz。当然音质越好占用的资源就越多,网络下载时间就越长。对于面向网络的 Applet 来说,必须考虑声音文件的大小,需要在音质和文件大小之间采取折衷办法。

在 Applet 中播放声音十分简单,加载声音文件,然后调用 play 方法播放即可。Java 提供了两种播放声音的方式:一种是通过 Applet 类的 play 方法,一种是通过 AudioClip 接口中的方法来播放。Applet 的 play 方法可以将声音文件的加载与播放一并完成,其调用格式如下:

```
void play(URL url)
void play(URL url, String name)
```

其中 URL 是一个网络地址,网络地址若包含声音文件可采用第一种形式,否则采用第二种形式,播放本地计算机上的声音文件也可采用第二种形式。假设有一个 MIDI 声音文件 trip. mid 和 Applet 放在同一个目录下,采用如下调用格式即可播放:

```
play(getCodeBase(), "trip. mid");
```

一旦 play 方法加载了该声音文件,就立即播放。如果找不到指定的声音文件,不会产生异常,只是听不到声音而已。然而,这种播放是一次性的,若要重播必须重新加载声音文件。如果你想把一个声音文件作为背景音乐连续播放,就需要引入功能更强的 AudioClip 接口,它包含在 java. applet 类库中,程序应引入 import java. applet. AudioClip。

声音文件的加载在创建 AudioClip 对象的过程中实现。Applet 的 getAudioClip 方法可创建这样的对象,该方法加载指定网络地址的声音文件,并返回一个 AudioClip 对象,调用格式如下:

```
AudioClip getAudioClip(URL url)
AudioClip getAudioClip(URL url, String name)
```

创建 AudioClip 对象后,声音文件即被加载,可调用它的方法处理声音文件。如果该方法没有找到指定的声音文件,将返回 null 值,此时不能引用所创建的对象。AudioClip 只有 3 个方法:

```
void play() 播放一遍;
void loop() 连续播放;
void stop() 停止播放。
```

如果只想播放一次,可调用 play 方法,也可在程序中通过循环调用 play 方法来控制播放次数。loop 方法可实现连续播放,但无法控制播放次数。stop 方法用来停止正在播放的声音剪辑。如果需要的话,可以加载多个声音文件一起播放,这些声音将混合起来,就像多重奏一样。

### 11.3.2 在 Applet 中播放声音

例 11.11 本地计算机工作目录下有 5 个声音文件,代表了 5 种声音格式,加载并播放这些声音文件。结果如图 11.10 所示。

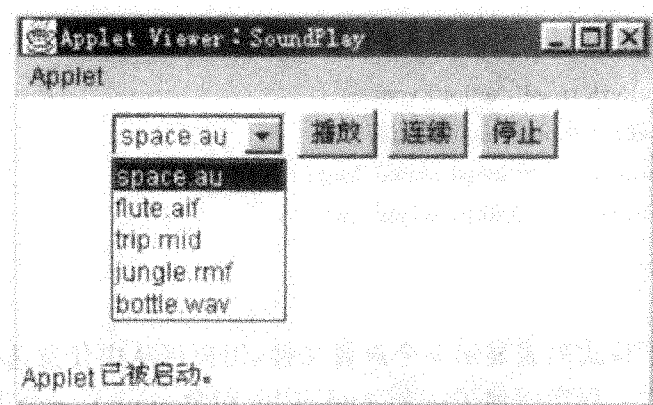


图 11.10

```
import java. awt. * ;
import java. awt. event. * ;
import java. applet. Applet;
```

```

import java.applet. AudioClip;

public class SoundPlay extends Applet implements ItemListener, ActionListener {
    AudioClip sound;
    Choice c=new Choice();
    Button play=new Button("播放");
    Button loop=new Button("连续");
    Button stop=new Button("停止");

    public void init() {
        c.add("space.au");
        c.add("flute.aif");
        c.add("trip.mid");
        c.add("jungle.rmfl");
        c.add("bottle.wav");
        add(c);
        c.addItemListener(this);
        add(play);
        add(loop);
        add(stop);
        play.addActionListener(this);
        loop.addActionListener(this);
        stop.addActionListener(this);
        sound=getAudioClip(getCodeBase(), "space.au");
    }

    public void itemStateChanged(ItemEvent e) {
        sound.stop();
        sound=getAudioClip(getCodeBase(), c.getSelectedItem());
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == play) sound.play();
        else if (e.getSource() == loop) sound.loop();
        else if (e.getSource() == stop) sound.stop();
    }
}

```

上述程序使用下拉式列表显示 5 个声音文件,用户可从中任意选择一个来播放,并用按钮来控制播放方式。程序添加了一个 AudioClip 对象 sound,一个下拉式列表 c 和 3 个按钮,实现了选项事件监听接口和动作事件监听接口。

在 init 方法中,添加了全部子对象,并注册了各自的事件监听者,最后加载了第一个声音文件 space.au,将对象引用赋给 sound。在选项事件处理方法中,首先停止当前声音文件的播放,然后根据用户的选择加载声音文件,并将对象引用赋值给 sound。

在按钮事件处理方法中,首先取得事件源,然后根据事件源的判断结果分别调用 sound 对象的相应方法。点击“播放”按钮时,调用 play 方法播放用户选择的声音文件,播放完毕就自动停止。点击“连续”按钮时,调用 loop 方法连续播放用户选择的声音文件,此方法不能自动停止。点击“停止”按钮时,调用 stop 方法停止播放。

此外,用户可以从下拉式列表中多次选择声音文件,点击“播放”或“重复”按钮就可实现混合播放。

## 习 题

- 11-1 如何加载和显示一幅图像? 如何放大和缩小一幅图像的显示?
- 11-2 如何生成一幅内存图像? 如何消除画面切换时的闪烁?
- 11-3 何为图形双缓冲技术? 编写动画程序时需要注意哪些因素?
- 11-4 如何加载和播放声音文件?
- 11-5 编写一个 Applet,显示一幅图像并配上背景音乐。
- 11-6 编写一个 Applet,动画显示一个数码钟,用数字显示时、分、秒。
- 11-7 编写一个 Applet,动画显示圆形时钟,显示时针和分针。
- 11-8 编写一个 Applet,实现向上滚动显示一行文字。
- 11-9 编写一个 Applet,实现向左滚动显示一行文字。
- 11-10 编写一个 Applet,实现向上滚动显示一幅图像。
- 11-11 编写一个 Applet,实现网页上的 banner 技术即连续显示多幅图像,每秒钟显示 6 幅图像。
- 11-12 编写一个 Applet,实现拉幕特效,即将一幅图像从中间向左右两边逐渐显示出来。
- 11-13 编写一个 Applet,实现一行文字蛇行移动。
- 11-14 编写一个 Applet,实现一个小球沿正弦曲线移动。
- 11-15 编写一个 Applet,实现一个小球沿抛物线移动,到达地面时播放一个响声。



# 第12章

## 流、多线程和网络编程

本章包括 Java 的流处理、多线程和网络编程,这部分内容对于深入了解 Java 语言是十分必要的。尽管从字面上看这些内容好像很深奥,但实际上它们还是体现了 Java 简单易学的特点,读者可以通过大量浅显的例子很快掌握这些内容。

### 12.1 流处理

一个程序在运行时通常要和外部交互,即从外部获取信息或向外部发送信息,这就是所谓的输入/输出(I/O)。信息可以来自不同地方:一个文件、磁盘、内存、另一个程序或是网络。也可以有不同的格式:字符串、图像、声音或对象。

在 Java 程序中,为了输入/输出信息,将在程序和信息发送者或接收者之间建立一个数据通道,以流(Stream)的形式来表示。输入就是读取数据流,输出则是写入数据流。不管信息的收发地点在何处,也不管信息是什么格式,程序都以流的形式来统一处理。处理过程也完全相同:打开流、读取或写入信息、关闭流。这样的设计,使 Java 程序在处理不同设备的 I/O 时非常方便。

#### 12.1.1 流

我们先来回顾一下第 9 章的例 9.10,在它的事件处理方法中有文件输入/输出流操作,可以打开文件和保存文件。下面是两个有关文件流处理的程序片断:

```
// 读文件
try {
    FileInputStream in=new FileInputStream(fileName); // 建立文件输入流
    in.read(byteBuf);           // 将文件内容读到字节数组
    in.close();                 // 关闭文件输入流
    str=new String(byteBuf);     // 将字节数组转换成字符串
    ...
} catch (IOException ioe) {}

// 写文件
```

```

str=ta.getText();           // 将文本区内容读至字符串
byteBuf=str.getBytes();     // 将字符串转换成字节数组
try {
    FileOutputStream out=new FileOutputStream(fileName); // 建立文件输出流
    out.write(byteBuf);      // 将字节数组写入文件输出流
    out.close();             // 关闭文件输出流
} catch (IOException ioe) {}

```

我们看到读文件时,程序以文件名为参数建立了一个输入流对象,调用它的 read 方法读取信息。这里是一次性读取,读完文件后,调用 close 方法关闭这个输入流。

写文件时,程序以文件名为参数建立了一个输出流对象,调用它的 write 方法写文件内容。这里是一次性写入,写完文件后,调用 close 方法关闭这个输出流。

文件流是基于二进制字节的,而文本区只能显示字符串,二者是不兼容的。可以利用下面介绍的派生流类进行转换。但上面的两个程序片断没有采用更多的派生类,而是利用字节数组和字符串进行相互转换,这样也很方便,唯一的限制是字节数组要足够大。至此,你对 Java 的流有了初步印象。下面我们进一步介绍流的概念。

Java 程序不能直接操纵 I/O 设备,而是在程序和设备之间加入了一个中介介质,这就是流。流是数据传输的抽象表达,与具体设备无关。程序一旦建立了流,就可以不用理会起点或终点是何种设备。

建立流实际上就是建立数据传输通道,将起点和终点连接起来。例如,程序要读写文件,你可以在程序和文件之间建立一个流。如果要从文件中读数据,则文件是起点,程序是终点;如果要将数据写入文件,则刚好相反。显然,数据流应分为输入流和输出流。

## 12.1.2 流的分类

java.io 包封装了大量的数据流类,它们以数据类型不同而分为两个继承层次:基于字节流的数据流类和基于字符流的数据流类。

基本流类有 4 个,它们都是抽象类:基于 Unicode 字符的输入流 Reader 和输出流 Writer,基于二进制字节的输入流 InputStream 和输出流 OutputStream,见表 12.1 至表 12.8。其他所有数据流类都是从它们中派生出来的。

表 12.1 Reader 的子类

名 称	功 能
BufferedReader	读取输入流到缓冲区
CharArrayReader	读取输入流到内建字符数组
FilterReader	过滤输入流的抽象类
InputStreamReader	读取字节流并转换为字符流
PipedReader	建立输入流管道连接到输出流
StringReader	建立数据源为字符串的输入流



表 12.2 Writer 的子类

名 称	功 能
BufferedWriter	将字符数据写入缓冲区
CharArrayWriter	将字符数据写入输出流缓冲区
FilterWriter	过滤输出流的抽象类
OutputStreamWriter	将字符流转换为字节流输出
PipedWriter	建立输出流管道连接到输入流
PrintWriter	将格式化对象写入文本输出流
StringWriter	建立终点为字符串的输出流

表 12.3 InputStream 的子类

名 称	功 能
AudioInputStream	读取声音字节流
ByteArrayInputStream	读取输入流到内存缓冲区
FileInputStream	读取文件输入流
FilterInputStream	建立可过滤的输入流
ObjectInputStream	读取对象并还原,如图像
PipedInputStream	建立输入流管道连接到输出流
SequenceInputStream	建立顺序输入流并逐个读取
StringBufferInputStream	JDK 1.3 不再支持

表 12.4 OutputStream 的子类

名 称	功 能
ByteArrayOutputStream	将字节数据写入缓冲区
FileOutputStream	写入文件输出流
filterOutputStream	建立可过滤的输出流
ObjectOutputStream	将对象原始数据类型写入输出流
PipedOutputStream	建立输出流管道连接到输入流

这些流类以及它们的子类在创建以后将被自动打开,可以调用 close 方法关闭它们。也可以交给垃圾收集器处理,当不再引用这些对象时,垃圾收集器会自动关闭它们。

表 12.5 InputStream 的主要方法

名 称	功 能
int read()	读取输入流的下一个字节
int read(byte b[])	将输入流读到字节数组中
int read(byte b[], int off, int len)	从输入流向字节数组的 off 处读取 len 个字节
long skip(long n)	从输入流中跳过 n 个字节
abstract void close()	关闭输入流释放资源

表 12.6 OutputStream 的主要方法

名 称	功 能
void write(int b)	将整型数 b 的低 8 位写入输出流
void write(byte b[])	将字节数组写入输出流
void write(byte b[], int off, int len)	从字节数组的 off 处向输出流写入 len 个字节
abstract void flush()	强制将输出流保存在缓冲区中的数据写入终点
abstract void close()	先调用 flush 然后关闭输出流释放资源

表 12.7 Reader 的主要方法

名 称	功 能
int read()	读取输入流的下一个字符
int read(char ch[])	将输入流读到字符数组中
int read(char ch[], int off, int len)	从输入流向字符数组的 off 处读取 len 个字符
long skip(long n)	从输入流中跳过 n 个字符
abstract void close()	关闭输入流释放资源

表 12.8 Writer 的主要方法

名 称	功 能
void write(int c)	将整型数 c 的低 16 位写入输出流
void write(char ch[])	将字符数组写入输出流
void write(char ch[], int off, int len)	从字符数组的 off 处向输出流写入 len 个字符
void write(String str)	将字符串写入输出流
void write(String str, int off, int len)	从字符串的 off 处向输出流写入 len 个字符
abstract void flush()	强制将输出流保存在缓冲区中的数据写入终点
abstract void close()	先调用 flush 然后关闭输出流释放资源

### 12.1.3 数据流的应用

下面,我们通过例子介绍常用输入/输出流的基本用法。对新出现的子类,也简要介绍了它们的构造方法和常用的成员方法。

#### 1. 文件输入/输出流的应用

**例 12.1** 将一个文件复制到另一个文件中,使用基于字节的文件输入/输出流。

```
import java.io.*;

public class BytesCopy {

    252 •
```

```

public static void main(String[] args) throws IOException {
    FileInputStream in=new FileInputStream("BytesCopy.java");
    FileOutputStream out=new FileOutputStream("BytesCopy.txt");
    int c;
    while ((c=in.read())!= -1)
        out.write(c);
    in.close();
    out.close();
}
}

```

这是一个非常简单的程序,我们建立了文件输入流 in 和文件输出流 out。使用循环重复读取输入流字节放入变量 c 中,注意变量类型是整型,因为 read 方法的返回值是整型,字节被加上高 24 位(3 个字节)成为整型。

调用输出流的 write 方法,将 c 写入输出流即新文件中,注意这里又将整型变量转为字节写入。当文件读完后,read 方法将返回 -1,于是退出循环并关闭输入流和输出流。

这个程序在运行时没有屏幕输出,但会生成新文件。此外,它采用字节读写,可以用来复制任何类型的文件,相当于 DOS 的 Copy 命令。

文件输入/输出流只能读写字节流,有时使用起来很不方便。Java 提供了基于字符流的文件读写器: FileReader 和 FileWriter,它们分别是 InputStreamReader 和 OutputStreamWriter 的子类。我们看下面的例子。

**例 12.2** 将一个文件复制到另一个文件中,使用基于 Unicode 字符的文件读写器。

```

import java.io.*;
public class Copy {
    public static void main(String[] args) throws IOException {
        FileReader in=new FileReader("Copy.java");
        FileWriter out=new FileWriter("Copy.txt");
        int c;
        while ((c=in.read())!= -1)
            out.write(c);
        in.close();
        out.close();
    }
}

```

本例与上例的功能和效果完全一样,不同的是这里使用了基于 Unicode 字符的文件读写器,每次可以处理一个 16 位的字符。

这里,我们概括出文件输入/输出流和文件读写器的用法:

- (1) 创建文件流对象,使它们连接到文件上,操作对象实际上就是操作文件。
- (2) 调用对象的方法处理数据,可以读或写数据。不同的读写方法得到的数据格式是不同的,这一点要特别注意。
- (3) 数据处理完毕最好调用 close 方法关闭文件流,使数据真正写入文件。

有关文件的流类是直接继承下来的,对父类的改变很少,它们继承了父类的读写方法并添加了构造方法和少量的成员方法,见表 12.9。

表 12.9 文件流类的构造方法

名 称	说 明
FileInputStream (String name)	name 代表文件名,可包含路径
FileInputStream (File file)	file 是文件类对象,代表一个指定文件
FileInputStream (FileDescriptor fdobj)	fdobj 代表一个打开的 I/O 设备
FileOutputStream (String name)	name 代表文件名,可包含路径
FileOutputStream (String name, boolean a)	a 取真值则将数据添加在文件尾部
FileOutputStream (File file)	file 是文件类对象,代表一个指定文件
FileOutputStream (FileDescriptor fdobj)	fdobj 代表一个打开的 I/O 设备

FileReader 和 FileWriter 的构造方法和上述构造方法分别对应,参数完全相同。

文件流类仅具备基本的字节或字符顺序读写功能,所以它们经常作为基本数据源和其他流类配合使用,以便利用更多的数据处理方法。

## 2. 缓冲流的应用

例 12.3 将文件输入流作为缓冲流的数据源,显示文件内容。结果如图 12.1 所示。

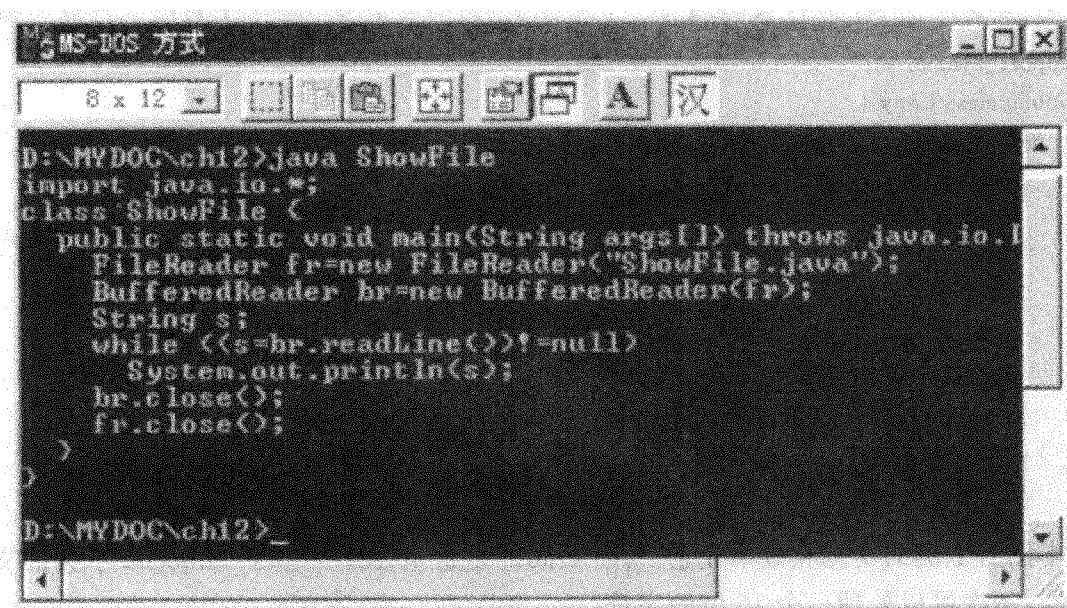


图 12.1 将文件输入流作为缓冲流的数据源,显示文件内容

```
import java.io.*;
class ShowFile {
    public static void main(String args[]) throws java.io.IOException {
        FileReader fr=new FileReader("ShowFile.java");
```

```

        BufferedReader br=new BufferedReader(fr);
        String str;
        while ((str=br.readLine())!=null)
            System.out.println(str);
        br.close();
        fr.close();
    }
}

```

在这个程序中,先用文件读取器创建了文件输入流对象 fr,然后将它作为缓冲读取器的参数创建缓冲读取器对象 br。执行时,从 fr 读取的内容将保存在 br 的缓冲区内。程序使用一个循环从 br 的缓冲区逐行读取数据到字符串 str 中,然后显示在屏幕上。最后,关闭这两个输入流。缓冲读取器提供了一个新的方法 readLine,可一次读取一行内容,对于屏幕显示或文本区显示十分有用。

BufferedReader 有一个缓冲区,它将尽可能多地读取数据到缓冲区内。BufferedWriter 同样也是尽可能多地将数据写入自己的缓冲区内,然后集中发送到终点,以减少系统和外部设备的交互次数。它们各有两个构造方法,见表 12.10。

表 12.10 缓冲读写器的构造方法

名 称	说 明
BufferedReader(Reader in)	in 是读取器对象,缓冲区为默认大小
BufferedReader(Reader in, int size)	in 同上,size 指定缓冲区大小
BufferedWriter(Writer out)	out 是写入器对象,缓冲区为默认大小
BufferedWriter(Writer out, int size)	out 同上,size 指定缓冲区大小

BufferedWriter 新增了一个成员方法 newLine,它可以产生一个换行符。

和缓冲读写器对应的是缓冲输入/输出流:BufferedInputStream 和 BufferedOutputStream。它们是从过滤流 FilterInputStream 和 FilterOutputStream 继承下来的。构造方法和缓冲读写器的构造方法类似,仅参数改为 InputStream 和 OutputStream。

例 12.4 使用缓冲输出流输出数据。结果如图 12.2 所示。

```

import java.io.*;
class BufferOut {
    public static void main(String args[]) throws IOException {
        BufferedInputStream in=new BufferedInputStream(System.in);
        byte b[]=new byte[10];
        System.out.println("\n 输入字符串,按回车键结束:");
        in.read(b, 0, 10);
        BufferedOutputStream out=new BufferedOutputStream(System.out);
        out.write(b, 0, 10);
    }
}

```

```

        System.out.println("\n 你输入的字符串是:");
        out.flush();
        System.out.println("\n");
    }
}

```

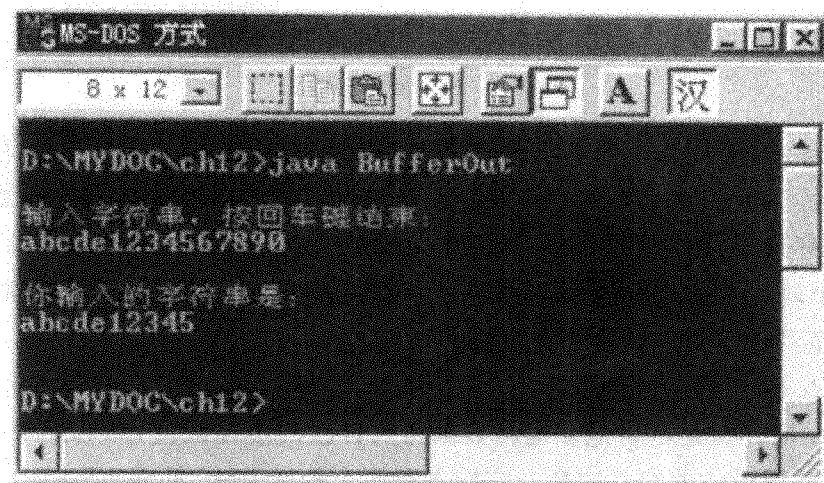


图 12.2

这个程序利用系统输入/输出流作为缓冲流的参数,从键盘读取字节流放入字节数组 `b` 中(最多容纳 10 个字节)。调用 `write` 方法把 `b` 写入输出缓冲区,最后调用 `flush` 方法输出这些数据,这里的数据终点是标准输出设备即屏幕。

### 3. 数据输入/输出流的应用

数据输入/输出流 `DataInputStream` 和 `DataOutputStream` 也是从过滤流 `FilterInputStream` 和 `FilterOutputStream` 继承下来的,在输入/输出的同时能对流进行变换处理,因此可以读写任意一种基本数据类型,如浮点数、整型数、字符等。

**例 12.5** 基本数据类型的输入/输出,这是一个买了东西计算总价款的例子,数据被存入一个文件。运行结果如图 12.3 所示。

```

import java.io.*;

public class DataTest {
    public static void main(String[] args) throws IOException {
        FileOutputStream fout=new FileOutputStream("data.txt");
        DataOutputStream out=new DataOutputStream(fout);
        float[] prices={1.5f,2.1f,2.9f,1.8f,3.1f};
        int[] units={5,2,4,3,1};
        String[] items={"苹果","鸭梨","蜜桃","橙子","葡萄"};

        for (int i=0; i<prices.length; i++) {
            out.writeFloat(prices[i]);
            out.writeChar('\t');

```

```

        out.writeInt(units[i]);
        out.writeChar('\t');
        out.writeChars(items[i]);
        out.writeChar('\n');
    }
    out.close();
    fout.close();

    FileInputStream fin=new FileInputStream("data.txt");
    DataInputStream in=new DataInputStream(fin);
    float price,total=0.0f;
    int unit;
    char ch;
    String item;

    try {
        while (true) {
            price=in.readFloat();
            in.readChar();          // 跳过 Tab
            unit=in.readInt();
            in.skip(2);              // 跳过 Tab
            item=new String();       // 清空字符串
            while((ch=in.readChar())!='\n') // 每次读一个字符,碰到换行符结束
                item+=ch;           // 将字符合成字符串
            System.out.println("你买了"+unit+"斤"+item+" 单价是"+price);
            total+=unit*price;
        }
    } catch (EOFException e) {}    // 读到文件尾将抛出异常并终止循环
    in.close();
    fin.close();
    System.out.println("你总共花去"+total+"元");
}
}

```

在这个例子中,数据输入/输出流建立在文件输入/输出流的基础上。数据输出流对象 out 以文件输出流对象 fout 为参数,共输出了 4 种类型的数据:float 型的 prices、char 型的“\t”和“\n”、int 型的 units、String 型的 items。它们构成了一行,以制表符 Tab 来分隔。

由于文件输出流只能处理字节流,所以 out 向 fout 传递数据的同时还要进行转换,将 prices 转换为 4 个字节,“\t”和“\n”转换为 2 个字节,units 转换为 4 个字节,items 的每个字符转换为 2 个字节。如果你打开 data.txt 文件,看到的是一堆乱码,这就是数据转换的结果。

虽然存入文件的是乱码,但只要调用对应的读取方法,这些数据就能被数据输入流正

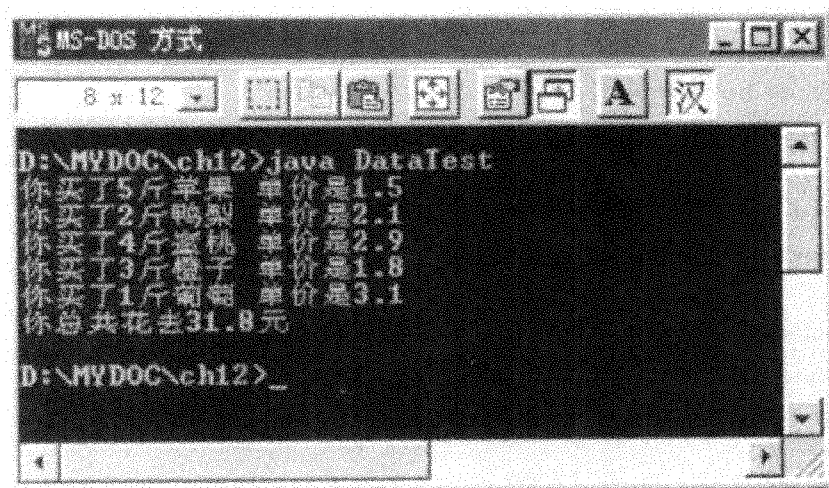


图 12.3

确还原。数据输入流对象 `in` 以文件输入流对象 `fin` 为参数,将 `fin` 传递过来的数据进行还原处理。读取一个浮点数存入 `price` 中,再读取一个字符,按照存入文件的顺序,这个字符是制表符而不需要保存。然后读取一个整型数存入 `unit` 中,再跳过一个字符,这里用 `skip` 方法跳过 2 个字节。

要特别注意字符串尤其是汉字字符串的读取,JDK1.3 版本不再支持老版本的一次读一行的方法 `readLine`,我们采用了先读字符后合成字符串的替代方法。首先将字符串 `item` 清空,在循环中读取一个字符放入 `ch` 中,然后将这个字符添加到 `item` 中,一旦读到换行符“`\n`”就退出循环。此时,文件中的一行就被读取并还原出来,可以在屏幕上显示了。

程序采用了一个条件恒为真的循环读取整个文件内容,那么怎样才能退出循环呢?显然不能用读取某个数据为退出条件,前面例子中有读取一个 `-1` 时结束循环的用法,这个方法在此处无效,因为文件中的数据很可能就包含 `-1`。但文件内容读完后再次试图读取,系统就会抛出一个 `EOFException` 异常,我们就用它来结束循环。

数据输入/输出流类除了继承父类的基本读写方法外,还添加了很多适用于各种数据类型的读写方法,下面我们给出它们的构造方法和常用成员方法,参见表 12.11。

#### 12.1.4 文件类

前面介绍的文件输入/输出流仅提供简单的文件 I/O。为了全面管理文件系统,Java 还提供了两个类:一般文件类 `File` 和随机存取文件类 `RandomAccessFile`。前者提供操作系统目录管理的功能,允许用户访问文件属性和路径等信息,后者用于文件的随机读写操作。

##### 1. 一般文件类

例 12.6 显示一个文件的有关信息,结果如图 12.4。

```
import java.io.*;
```



```

class DirFile {
    public static void main(String args[]) throws IOException {
        File file=new File("DirFile.java");

        if (file.exists()) {
            System.out.println("文件名   "+file.getName());
            System.out.println("路径     "+file.getPath());
            System.out.println("绝对路径:"+file.getAbsolutePath());
            System.out.println("文件长度:"+file.length()+" bytes");
        }
        else
            System.out.println("对不起,文件没找到。");
    }
}

```

表 12.11 数据输入/输出流类的常用方法

方 法	功 能
<code>DataInputStream(InputStream in)</code>	数据输入流的构造方法,in 为输入流对象
<code>int skipBytes(int n)</code>	跳过 n 个字节
<code>byte readByte()</code>	读取一个字节
<code>char readChar()</code>	读取一个字符
<code>int readInt()</code>	读取一个整型数
<code>float readFloat()</code>	读取一个浮点数
<code>double readDouble()</code>	读取一个双精度数
<code>boolean readBoolean()</code>	读取一个布尔值
<code>DataOutputStream(OutputStream out)</code>	数据输出流的构造方法,out 为输出流对象
<code>int size()</code>	返回写入的字节数
<code>void writeByte(int v)</code>	写入一个字节,忽略 v 的前 3 个字节
<code>void writeChar(int v)</code>	写入一个字符,忽略 v 的前 2 个字节
<code>void writeInt(int v)</code>	写入一个整型数,转换成 4 个字节
<code>void writeFloat(float v)</code>	写入一个浮点数,转换成 4 个字节
<code>void writeDouble(double v)</code>	写入一个双精度数,转换成 8 个字节
<code>void writeBoolean(boolean v)</code>	写入一个布尔值,转换成 1 个字节
<code>void writeBytes(String s)</code>	s 的每个字符被忽略高 8 位,以字节写入
<code>void writeChars(String s)</code>	s 的每个字符被转换成 2 个字节写入

这里 DirFile.java 就在当前目录,如果是显示其他目录中的文件,要在文件名前面加上路径,否则显示文件找不到。程序使用 File 生成一个文件对象 file,代表 DirFile.java。

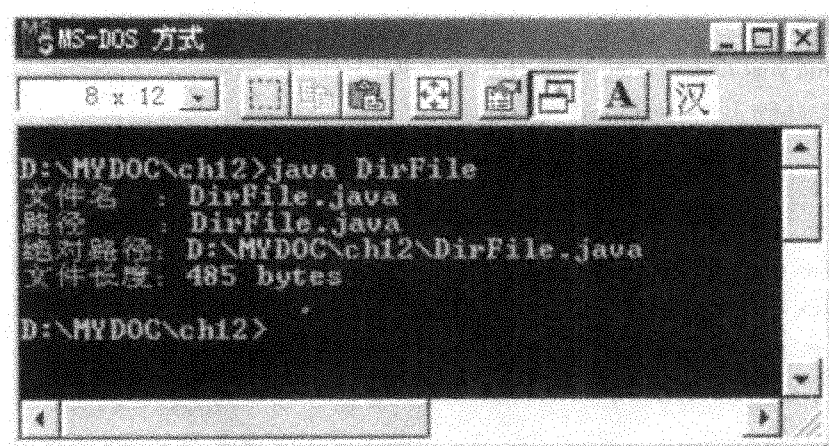


图 12.4

调用 `exists` 方法来确定这个文件是否存在, 如果存在, 则通过 `File` 的各种方法来获取文件信息并显示出来。 `File` 类的使用比较简单, 表 12.12 列出了它的主要方法。

表 12.12 `File` 的主要方法

方 法	功 能
<code>File(String path)</code>	构造方法, <code>path</code> 指定文件路径
<code>File(String path, String name)</code>	<code>path</code> 指定文件路径, <code>name</code> 指定文件名
<code>File(File path, String name)</code>	<code>path</code> 对象指定文件路径, <code>name</code> 指定文件名
<code>String getName()</code>	返回文件名
<code>String getPath()</code>	返回文件路径
<code>String getAbsolutePath()</code>	返回文件的绝对路径
<code>String getParent()</code>	返回文件的父目录
<code>boolean exists()</code>	返回一个确定文件是否存在的布尔值
<code>long length()</code>	返回文件的字节长度
<code>boolean mkdir()</code>	建立一个目录
<code>boolean delete()</code>	删除一个文件

## 2. 随机存取文件类

文件输入/输出流采用了顺序读写。有时需要有选择的读写文件, 例如读写一行或几行。 `RandomAccessFile` 可实现这种操作, 它能让你从文件的不同位置读写不同长度的数据。

**例 12.7** 在文件末尾添加字符串, 结果如图 12.5。

```
import java.io.*;
class AppendFile {
```

```

public static void main(String args[]) {
    String str[] = {"First line\n", "Second line\n", "Last line\n"};
    try {
        RandomAccessFile rf = new RandomAccessFile("demo.txt", "rw");
        System.out.println("\n 文件指针位置为:" + rf.getFilePointer());
        System.out.println("文件的长度为:" + rf.length());
        rf.seek(rf.length());
        System.out.println("文件指针现在的位置为:" + rf.getFilePointer());
        for (int i = 0; i < 3; i++)
            rf.writeBytes(str[i]); // 字符串转为字节串添加到文件末尾
        rf.seek(0);
        System.out.println("\n 文件现在内容:");
        String s;
        while ((s = rf.readLine()) != null)
            System.out.println(s);
        rf.close();
    }
    catch (FileNotFoundException fnoe) {}
    catch (IOException ioe) {}
}

```

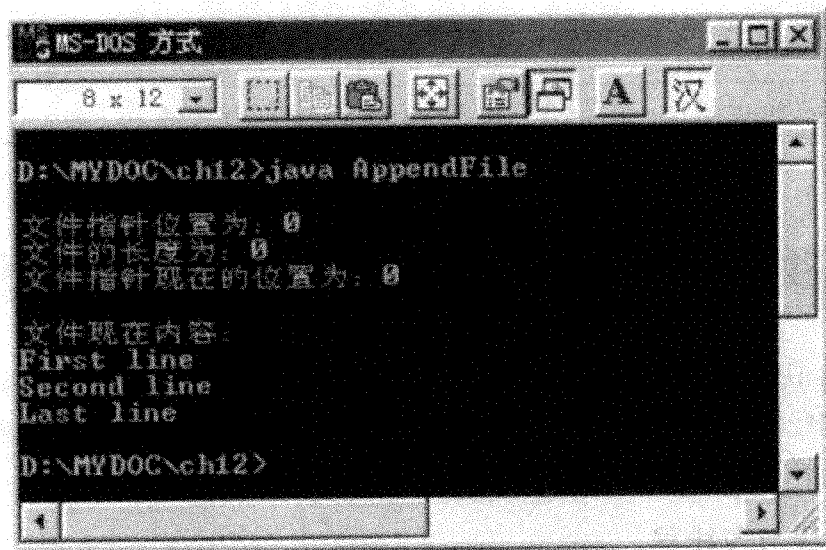


图 12.5

程序定义了一个字符串数组, 为它准备了 3 个字符串常量。创建了一个随机存取文件对象, 并以读写 rw 方式打开 demo.txt。seek 方法是很重要的一个方法, 可用它定位文件指针。例如 rf.seek(rf.length()) 就将文件指针移到了文件尾。

在循环中, 3 次调用 writeBytes 方法将字符串转为字节串后添加到文件指针下面。如果调用 writeChars 方法添加字符串, 将以 16 位的字符写入, 这样文件中每个字符的前面会多出一个空。字符转换为字节后, 只有低 8 位写入文件, 因此就不会出现空。如果程

序中有汉字,则必须使用 writeChars 方法,原因已在前面介绍过。最后,将指针移到文件头,调用 readLine 方法逐行取出文件内容显示在屏幕上。

RandomAccessFile 有很多方法,我们在表 12.13 列出了常用方法。

表 12.13 RandomAccessFile 的常用方法

方 法	功 能
RandomAccessFile(String name, String mode)	构造方法, name 指定文件名, mode 指定打开方式, r 为只读, rw 为读写
RandomAccessFile(File file, String mode)	file 对象指定文件名, mode 同上
long getFilePointer()	返回文件指针位置
void seek(long pos)	移动文件指针到指定位置
String readLine()	读取一行, 行以“\r”或“\n”为结束符
char readChar()	读取一个字符
byte readByte()	读取一个字节
int readInt()	读取一个整型数
void writeByte(int v)	写入一个字节, v 的低 8 位
void writeChar(int v)	写入一个字符, v 的低 16 位
void writeInt(int v)	写入一个整型数
void writeBytes(String s)	以字节序列写入字符串
void writeChars(String s)	以字符序列写入字符串

## 12.2 多线程

本节介绍 Java 语言的多线程特点。Java 在设计时就考虑了多线程问题,把它溶入到了 Java 编程中。通过学习,你可以很快掌握 Java 的多线程特点,编写多线程程序就像编写普通程序一样简单。

### 12.2.1 线程与多线程

程序员对编写序列化程序是非常熟悉的,无论是输出一个“你好”,还是对 10 个数进行排序,这些都是序列化程序。它们有一个开始、一个可执行的命令序列、一个结束。在程序执行的任何时刻,只有一个执行点。

线程(Thread)和序列化程序是类似的,也有开始、执行序列和结束,执行时也只有一个执行点。但线程不是程序,它不能自己运行,只能在程序中执行。我们称程序中单个序列化的流控制为线程。一个线程在程序运行时,必须争取到为自己分配的系统资源,如执行堆栈、程序计数器等。单个线程没有什么特别的意义。

多线程是相对于单线程而言的,指的是在一个程序中可以定义多个线程并同时运行它们,每个线程可以执行不同的任务。你可能对多任务有一定的体会,例如一边使用计算机编程,一边听计算机播放音乐。

多线程和多任务是两个既有联系又有区别的概念,多任务是针对操作系统而言的,代表着操作系统可以同时执行的程序个数;多线程是针对一个程序而言的,代表着一个程序内部可以同时执行的线程个数,而每个线程可以完成不同的任务。例如 Java 推出的 HotJava 浏览器,你可以一边浏览网页一边下载新网页,可以同时显示动画和播放音乐。

很多计算机编程语言需要利用外部软件包来实现多线程,而 Java 语言则内在支持多线程,所有的类都是在多线程思想下定义的。Java 的线程通过 java.lang 中的线程类 Thread 来实现,Thread 封装了所有需要的线程操作控制,有很多方法可以用来控制一个线程的运行、休眠、挂起或停止。

使用多线程编程可将程序任务分解为几个并行的子任务,通过线程的并发执行来加速程序运行,提高 CPU 的利用率。例如,在网络编程中,有很多功能可以并发执行。网络传输速度一般较慢,用户输入速度也较慢,你可以设计两个独立线程分别完成这两个任务而不影响正常的显示或其他功能。在编写动画程序时,你可以用一个线程进行延时,让另一个线程在延时中准备要显示的画面,以实现完美的动画显示。

### 12.2.2 创建线程

有两种方法可以创建线程。一种方法是通过继承线程类 Thread 来创建线程类;另一个方法是建立一个实现 Runnable 接口的类来运行线程。下面我们通过例子来说明这两种方法的使用。

**例 12.8** 通过继承 Thread 创建一个子类,在主控程序中同时运行两个线程。运行结果如图 12.6 所示。

```
class Thread1 {
    public static void main(String args[]) {
        testThread t1=new testThread("thread1");
        testThread t2=new testThread("thread2");
        t1.start();
        t2.start();
    }
}

class testThread extends Thread {
    public testThread(String str) {
        super(str); // 调用父类的构造方法为线程对象命名
    }

    public void run() {
        for (int i=0; i<3; i++) {
```

```

        System.out.println(getName()+"在运行");
    try {
        sleep(1000); // 用休眠 1000 毫秒来区分哪个线程在运行
    } catch (InterruptedException e) {}
}
System.out.println(getName()+"已结束");
}
}

```

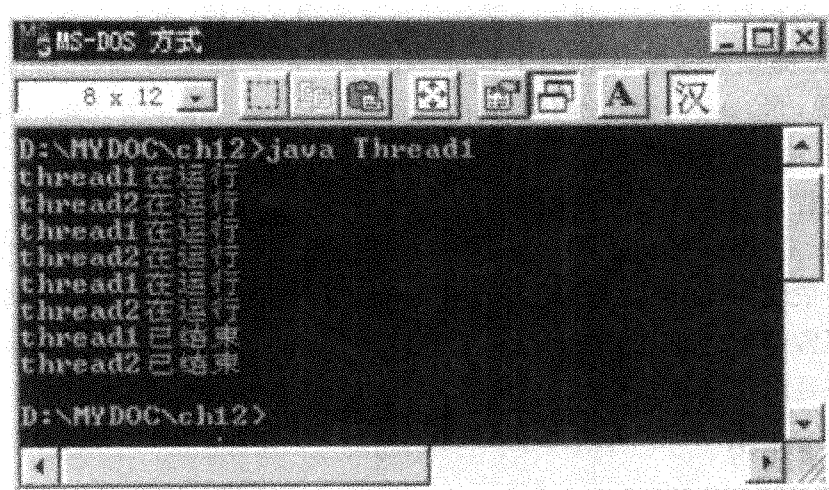


图 12.6 多线程运行结果

说明:

(1) 程序运行时总是调用 main 方法,因此 main 是创建和启动线程的地方。Thread1 的 main 方法中生成了两个 testThread 线程对象,一个为 thread1,一个为 thread2,并在创建后马上调用 start 方法启动这两个线程。

(2) 从输出的结果可以看出两个线程的名字是交替显示的,这是因为两个线程是同步的,于是,两个 run 方法也同时被执行。每一个线程运行到输出语句时将在屏幕上显示自己的名字,执行到 sleep 语句时将休眠 1000 毫秒。线程休眠时并不占用 CPU,其他线程可以继续运行。一旦延迟完毕,线程将被唤醒,继续执行下面的语句。这样,它们就实现了交替显示。

(3) Thread 创建的线程不做任何事情,因为它的 run 方法是空的。所以,对于继承自 Thread 的子类来说,你必须覆盖 run 方法。run 是线程类的关键方法,线程的所有活动都是通过它来实现的。当线程实例化后系统就自动调用 run 方法,正是通过 run 方法才使创建线程的目的得以实现。你可以在 run 方法里控制程序,一旦进入 run 方法,便可执行里面的任何语句,run 方法执行完毕,这个线程也就结束了。

**例 12.9** 通过 Runnable 接口运行线程,运行结果如图 12.7 所示。

```

import java.awt.*;
import java.applet.Applet;
import java.util.*;

```

```

import java, text, DateFormat;

public class Clock extends Applet implements Runnable {
    Thread clockThread=null;

    public void init() {
        setBackground(Color, blue);
        setForeground(Color, yellow);
    }

    public void start() {
        if (clockThread==null) {
            clockThread=new Thread(this,"Clock");
            clockThread.start();
        }
    }

    public void run() {
        Thread myThread=Thread.currentThread();
        while (clockThread==myThread) {
            repaint();
            try {
                Thread.sleep(1000);
            }
            catch (InterruptedException e) {}
        }
    }

    public void paint(Graphics g) {
        Date date=new Date();
        DateFormat formatter=DateFormat.getTimeInstance();
        String str=formatter.format(date);
        g.drawString(str,5,10);
    }

    public void stop() {
        clockThread=null;
    }
}

```

### 12.2.3 线程的生命周期

线程在它的生命周期内有 4 种状态:创建(New Thread)、运行(Runnable)、挂起(Not Runnable)、结束(Dead)。它们之间的关系如图 12.8 所示。

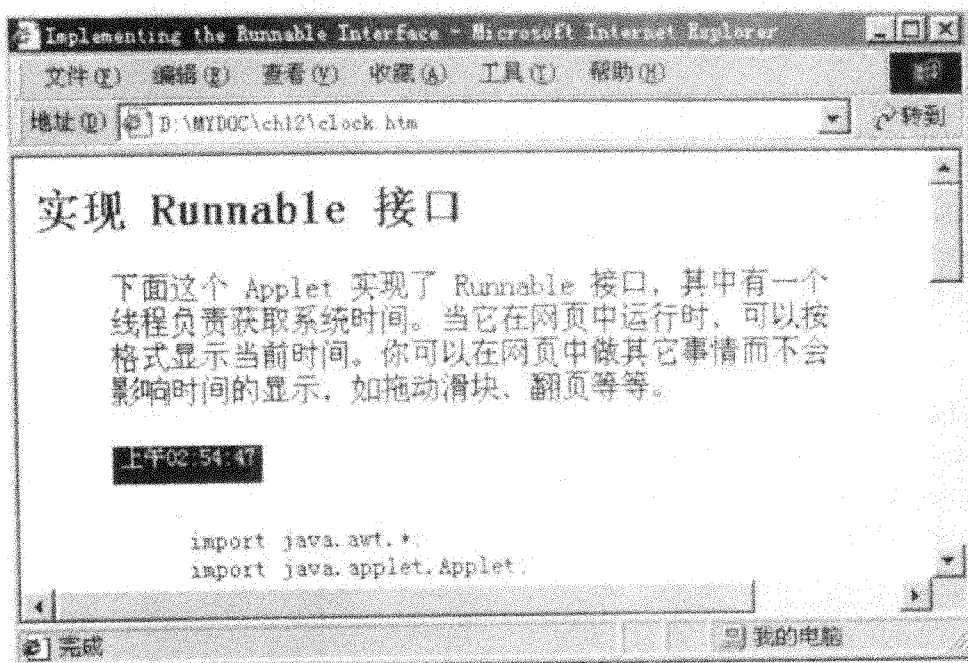


图 12.7

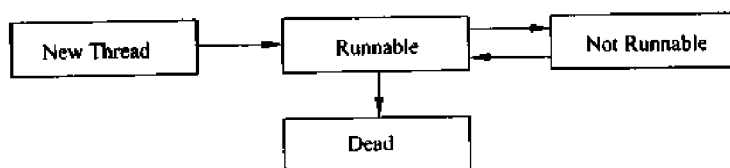


图 12.8

我们用例 12.9 说明线程的各个状态。当 Applet 被启动时,首先调用 Applet 的 `init` 方法设置背景和前景色,然后调用 `start` 方法。正是在 Applet 的 `start` 方法中用 `new` 操作符创建了一个线程:

```

public void start() {
    if (clockThread == null) {
        clockThread = new Thread(this, "Clock");
        clockThread.start();
    }
}

```

此时,线程 `clockThread` 处于 `New Thread` 状态,是一个空线程,因为还没有为它分配系统资源。处于 `New Thread` 状态的线程除了调用 `start` 方法开始线程外,不能调用其他方法,否则将产生 `IllegalThreadStateException` 异常。

调用 `start` 方法,将为线程分配系统资源,并将线程加入线程队列,然后调用 `run` 方法运行线程。此时,线程转入 `Runnable` 状态。由于大部分计算机只有一个 CPU,不可能同时运行所有处于 `Runnable` 状态的线程,于是 Java 虚拟机就建立了一个线程队列,让这些线程以排队的方式轮流使用 CPU。



```

public void run() {
    Thread myThread=Thread.currentThread();
    while (clockThread==myThread) {
        repaint();
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {}
    }
}

```

在 run 方法中,如果当前线程是 clockThread,就进入循环。在循环中先执行 repaint 方法,它将调用 paint 显示系统时间。然后令线程休眠 1000 毫秒,此时线程将停止运行。延时结束时线程被唤醒,如果 CPU 可用,就会继续运行这个线程,否则将排队等待。

以下 3 种情况会使线程转入 Not Runnable 状态:

- (1)调用 sleep 方法时,延时结束,重新转入 Runnable 状态。
- (2)调用 wait 方法等待一个特定状态发生时,状态发生后,其他对象必须调用 notify 或 notifyAll 方法向等待中的线程发出通知,才能唤醒这个线程。
- (3)当线程被 I/O 阻塞时,I/O 完成后可唤醒线程。

一般安排线程自然结束。在例 12.8 中,循环 3 次就自然结束线程。在例 12.9 中,进入循环的条件是 clockThread==myThread,怎样结束线程呢?如果你把这个 Applet 嵌入网页,当用户关闭这个网页时,Applet 的 stop 方法将被调用:

```

public void stop() {
    clockThread=null;
}

```

这里,clockThread 被赋值 null,破坏了循环条件,因此循环终止,自然结束线程。

线程的 isAlive 方法可返回一个线程的状态。如果返回值为真,说明线程处于 Runnable 或 Not Runnable 状态;如果返回值为假,说明线程处于 New Thread 或 Dead 状态。但无法进一步区分具体状态。

#### 12.2.4 线程的优先级

尽管从概念上可以说线程能同步运行,但事实上存在着差别。如果计算机有多个 CPU 则没有问题,但大部分计算机都是一个 CPU,一个时刻只能运行一个线程。单个 CPU 上运行多线程采用了线程队列技术,Java 虚拟机支持固定优先级队列,一个线程的执行顺序取决于它相对其他 Runnable 线程的优先级。

线程在创建时,继承了父类的优先级。线程创建后,你可以在任何时刻调用 setPriority 方法改变线程的优先级。优先级为 1~10,Thread 定义了其中 3 个常数:

- (1) MAX\_PRIORITY 最大优先级(值为 10);
- (2) MIN\_PRIORITY 最小优先级(值为 1);
- (3) NORM\_PRIORITY 默认优先级(值为 5)。

例 12.10 线程优先级的使用,运行结果如图 12.9 所示。

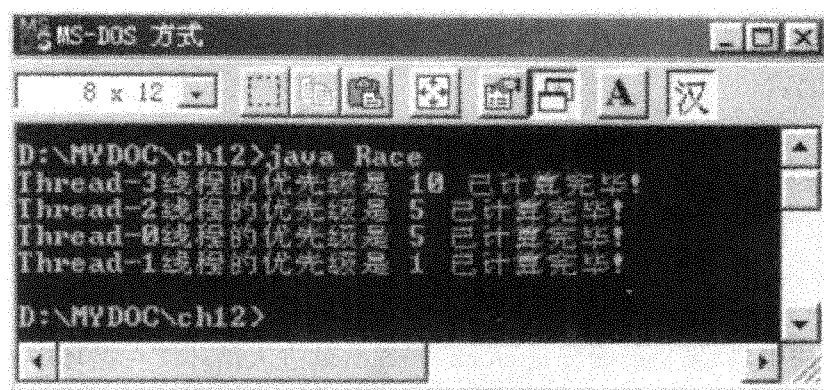


图 12.9

```
class Race extends Thread {
    public static void main(String args[]) {
        Race[] runner=new Race[4];
        for(int i=0;i<4;i++) runner[i]=new Race();
        for(int i=0;i<4;i++) runner[i].start();
        runner[1].setPriority(MIN_PRIORITY);
        runner[3].setPriority(MAX_PRIORITY);
    }

    public void run() {
        for(int i=0; i<1000000; i++);
        System.out.println(getName()+"线程的优先级是 "+getPriority()+" 已计算完毕!");
    }
}
```

程序创建了 4 个线程,其中第 2 个线程的优先级设为最小,第 4 个线程的优先级设为最大,其他两个为默认优先级。这些线程各自进行 1000000 次加法运算,循环结束后输出有关信息。每次执行的结果可能不相同,但第 4 个线程总是被最先执行。

### 12.2.5 线程同步

前面的线程例子都是独立的,而且异步执行,也就是说每个线程都包含了运行时所需要的数据和方法,不需要外部资源,也不用关心其他线程的状态和行为。但有时一些同时运行的线程需要共享数据,例如两个线程同时存取一个数据流,其中一个对数据进行了修改,而另一个线程使用的仍是原来的数据,这就带来了数据不一致问题。因此,编程时必须考虑其他线程的状态和行为,以解决资源共享问题。

Java 提供了同步设定功能。共享对象可将自己的成员方法定义为同步化(synchronized)方法,通过调用同步化方法来执行单一线程,其他线程则不能同时调用同一个对象的同步化方法。

生产者和消费者模型是典型的线程同步问题,下面我们通过这个模型来说明线程同

步的处理方法：

使用某种资源的线程称为消费者，产生或释放这个资源的线程称为生产者。生产者生成 10 个整数(0~9)，存储到一个共享对象中，并把它们打印出来。每生成一个数就随机休眠 0~100 毫秒，然后重复这个过程。一旦这 10 个数可以从共享对象中得到，消费者将尽可能快地消费这 10 个数，即把它们取出后打印出来。

**例 12.11** 生产者和消费者线程同步化问题。由 4 个程序组成，运行结果如图 12.10。



图 12.10

### (1) 生产者程序

```
public class Producer extends Thread {
    private Share shared;
    private int number;

    public Producer(Share s, int number) {
        shared=s;
        this.number=number;
    }

    public void run() {
        for (int i=0; i<10; i++) {
            shared.put(i);
            System.out.println("生产者"+this.number+" 输出的数据为："+i);
            try {
                sleep((int)(Math.random() * 100));
            } catch (InterruptedException e) {}
        }
    }
}
```

```
}  
}
```

## (2) 消费者程序

```
public class Consumer extends Thread {  
    private Share shared;  
    private int number;  
  
    public Consumer(Share s, int number) {  
        shared=s;  
        this.number=number;  
    }  
  
    public void run() {  
        int value = 0;  
        for (int i=0; i<10; i++) {  
            value=shared.get();  
            System.out.println("消费者"+this.number+" 得到的数据为:"+value);  
        }  
    }  
}
```

## (3) 共享资源对象

```
public class Share {  
    private int contents;  
    public int get(){  
        return contents;  
    }  
  
    public void put(int value){  
        contents=value;  
    }  
}
```

## (4) 主程序

```
public class PCTest {  
    public static void main(String[] args) {  
        Share s=new Share();  
        Producer p=new Producer(s,1);  
        Consumer c=new Consumer(s,1);  
        p.start();  
        c.start();  
    }  
}
```

在这个模型中,生产者向共享对象 Share 存入数据,消费者从 Share 中取出数据。但是你可以从运行结果中看出,程序无法保证生产者线程存入一个数据消费者线程就可以取出一个数据。原因是 Share 中的 put 和 get 方法没有实现同步化。

我们分析一下可能发生的情况:一种情况是生产者比消费者速度快,那么在消费者还没有取出上一个数据之前,生产者又存入了新数据,于是,消费者很可能会跳过上一个数据。另一种情况则相反,当消费者比生产者速度快,消费者可能两次取出同一个数据。

这两种情况不是我们所希望的。我们希望生产者存入一个数,消费者取出的就是这个数。为了避免上述情况发生,就必须锁定生产者线程,当它向共享对象中存储数据时禁止消费者线程从中取出数据,反之也一样。将共享对象 Share 中的 put 和 get 分别定义为同步化方法就可达到这个目的。

(5) 共享资源对象实现同步化,运行结果如图 12.11。

图 12.11

```
public class Share {
    private int contents;
    private boolean available=false;

    public synchronized int get() {
        while (available==false) {
            try {
                wait();
            } catch (InterruptedException e) {}
        }
        available=false;
        notifyAll();
        return contents;
    }
}
```

```

public synchronized void put(int value) {
    while (available == true) {
        try {
            wait();
        } catch (InterruptedException e) { }
    }
    contents = value;
    available = true;
    notifyAll();
}
}

```

修改后的 Share 仍利用 put 和 get 方法来写入和读取数据,但增加了 wait 和 notifyAll 功能。wait 使线程进入短暂休眠,收到 notifyAll 的通知后会马上醒来。当消费者线程调用共享对象的 get 方法时,如果生产者没有写入数据,available 变量就会保持为假,线程进入循环并调用 wait 方法等待。一旦生产者写入了新数据,available 的值就会改变,同时生产者还会向消费者发出通知,唤醒消费者线程退出循环。此时,消费者线程将做两个非常重要的工作,一是把 available 变量改为假,二是通知生产者线程。最后,返回 contents,它包含最新写入的数据。

当生产者线程第一次调用共享对象的 put 方法时,available 变量为假,线程将跳过循环并将第一个数据写入 contents 变量,然后将 available 变量改为真值,调用 notifyAll 方法通知消费者线程可以取数据了。再次调用 put 方法时,如果消费者没有取走数据,available 变量就会保持为真,线程将进入循环并调用 wait 方法等待。一旦消费者取走上一个数据,available 的值就会改变,线程也会被唤醒并退出循环,继续后面的工作。采用这样的处理方式,就可以保证消费者一直等到生产者写入一个新数据后再把它取出,而生产者则一直等到消费者取走上一个数据后再写入新数据。

## 12.2.6 多线程问题

没有任何事情是完美的,多线程也不例外,你应该清醒地意识到在程序中使用多线程是有代价的。首先,对于程序员来说,必须加倍注意自己的多线程程序。由于多线程实际上是多个程序段同时运行于内存中,所以一定要理清它们的关系,不要让它们搅乱了你的头脑。如果真的出现这种情况,那么最好还是少用几个线程。其实,并不是线程越多程序就执行得越快,还有很多其他因素决定着程序的执行速度。

其次,对多线程程序本身来说,它会对系统产生以下影响:

- (1) 线程需要占用内存。
- (2) 线程过多,会消耗大量 CPU 时间来跟踪线程。
- (3) 必须考虑多线程同时访问共享资源的问题,如果没有协调好,就会产生令人意想不到的问题,例如可怕的死锁和资源竞争。
- (4) 因为同一个任务的所有线程都共享相同的地址空间,并共享任务的全局变量,所以程序也必须考虑多线程同时访问全局变量的问题。

## 12.3 网络编程

Internet 越来越成为人们生活中不可缺少的内容了,一旦接触了它,你就很难拒绝它的诱惑。从程序员的角度来看,面向 Internet 的编程将会是今后的主要任务,而 Java 正好提供了这样的编程利器。从设计之初,Java 就把网络功能作为一个内置的基本功能,访问网络资源就像访问本地资源一样方便。

### 12.3.1 Java 网络基础知识

Java 获得迅速推广的一个重要原因就在于强大的网络能力,利用网络类,Java 程序能够方便地访问 Internet 和 World Wide Web 上的资源。接入 Internet 的计算机相互之间的通信要么采用 TCP 协议要么采用 UDP 协议,这些是较低层次的协议。当你编写 Java 网络程序时,是基于应用层的,不需要考虑那些复杂的底层协议。java.net 类库包含了你所需要的各种网络类,使用它们就可以编写出基于 TCP 或 UDP 协议的网络程序。但为了正确使用网络类,你还是应该了解 TCP 和 UDP 的区别。

传输控制协议 TCP(transmission control protocol)是一个基于连接的协议,可在两台相连计算机之间提供可靠的数据流。当两台计算机要传输信息时,需要建立一个可靠的连接以便发送和接收。这个过程和两个人打电话类似,TCP 就像电话局一样,保证一方发出的数据按顺序到达另一方。目前流行的 HTTP、FTP、Telnet 都是基于这种点对点通信的 TCP 协议。

用户数据报协议 UDP(user datagram protocol)是一个无连接的、发送独立数据包(即数据报)的协议,它不保证数据的正确到达。这个过程和通过邮局发信类似,每封信的内容是独立的,既不保证收信方按顺序收到信件,也不保证收信方一定能收到信件。

一般来讲,计算机只有一个物理通道连接到网络,所有数据都由这个通道进出。但这些数据可能是计算机上的不同应用程序所需要的,怎样区别它们呢?用端口区分。Internet 传输的数据都带有地址,包含了计算机和端口信息。计算机以 32-bit IP 地址表示,端口用一个 16-bit 数表示,TCP 和 UDP 就是使用了这个端口数才能把数据发给正确的程序。基于连接的 TCP 协议,服务器会把一个套接字(Socket)和一个指定端口绑在一起,客户机连接到这个端口就可接收所有指向这个端口的数据。而基于数据报的 UDP 协议,每个数据包都包含端口数,UDP 可以指引它们到达正确的应用程序。

很多应用程序需要可靠的、按顺序的数据传输,也有的应用程序不需要,因此在设计网络程序时应正确选择网络类。URL、URLConnection、Socket 和 ServerSocket 类是基于 TCP 协议的,而 DatagramPacket、DatagramSocket 和 MulticastSocket 类是基于 UDP 协议的。

### 12.3.2 URL 编程

URL(统一资源定位符)代表着 Internet 上的指定资源。为浏览器指定了 URL 你就

可以访问这个资源。Java 程序同样可以使用 URL 访问网络资源,java.net 提供了 URL 类,可用它创建一个代表 URL 地址的对象,通过对象就可以访问指定资源。

**例 12.12** 利用 URL 访问网站,如图 12.12 所示。

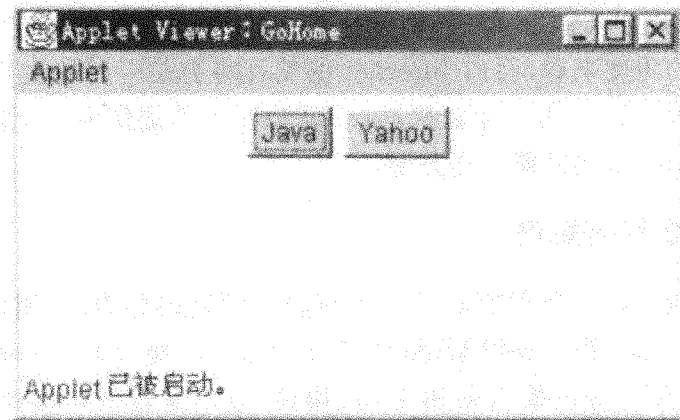


图 12.12

```
import java.awt.* ;
import java.awt.event.* ;
import java.net.* ;
import java.applet.* ;

class HomeButton extends Button {
    String name;
    URL home;
    HomeButton(String name, String site) {
        setLabel(name);
        try {
            home=new URL(site);
        }
        catch (MalformedURLException e) {}
    }
}

public class GoHome extends Applet implements ActionListener {
    HomeButton javaHome=new HomeButton("Java", "http://java.sun.com");
    HomeButton yahooHome=new HomeButton("Yahoo", "http://www.yahoo.com");

    public void init() {
        add(javaHome);
        add(yahooHome);
        javaHome.addActionListener(this);
        yahooHome.addActionListener(this);
    }
}
```



```

public void actionPerformed(ActionEvent e) {
    HomeButton btn=(HomeButton)e.getSource();
    getAppletContext().showDocument(btn.home);
}

```

这个程序可以访问两个网站:Java 和 Yahoo,点击按钮就可打开相应网站的首页。但在 Applet 查看器中仅仅显示了 Applet 的外貌,只有在浏览器中才能真正打开网页。

HomeButton 是按钮的子类,增加了两个属性,name 用来设置按钮名称,home 是一个 URL 对象。创建 URL 对象时会抛出 MalformedURLException 异常,因此使用了 try...catch 结构,根据传递的参数来创建这个 URL 对象。

主类创建了两个 URL 对象,一个是网站的名称和对应的网址。这里的网址是一个字符串,包含了两部分内容,http 表示应用层协议,双斜杠后面是主机名,例如 java.sun.com 代表 Sun 公司的网络服务器主机,程序没有给出文件名,浏览器将使用默认文件名即 index.html。

在 init 方法中,将两个按钮添加到 Applet,注册于它们的事件监听者。在事件处理方法中,btn 代表所点击的按钮。点击按钮,浏览器将启动拨号程序访问按钮代表的网站。

注意 Applet 方法的使用:getAppletContext 方法的返回值是一个对象引用,这个对象引用就是运行 Applet 的浏览器。showDocument 方法需要一个 URL 对象,这里用 btn.home 作为参数,它就是在 HomeButton 类中创建的 URL 对象。

### 12.3.3 创建 URL 对象

从上面的例子来看,URL 编程的关键是 URL 对象的创建。下面是 URL 对象的构造方法,根据参数的不同,可以创建多种形式的 URL 对象:

(1) URL(String url) url 代表一个绝对地址,URL 对象直接指向这个资源。上面的例子就采用了这个构造方法创建 URL 对象。

(2) URL(URL baseUrl, String relativeURL) baseUrl 代表绝对地址,relativeURL 代表相对地址。

例如:URL gamelan=new URL("http://www.gamelan.com/pages/");

URL gamelanGames=new URL(gamelan, "Gamelan.game.html");

则 gamelanGames 指向 http://www.gamelan.com/pages/Gamelan.game.html。

(3) URL(String protocol, String host, String file) protocol 代表通信协议,host 代表主机名,file 代表文件名。

例如:new URL("http", "www.gamelan.com", "/pages/Gamelan.net.html");

等价于:new URL("http://www.gamelan.com/pages/Gamelan.net.html");

(4) URL(String protocol, String host, int port, String file) protocol 代表通信协议,host 代表主机名,port 代表端口,file 代表文件名。

例如: `URL url = new URL("http","www.gamelan.com", 80, "pages/Gamelan.network.html");`

对象 `url` 则指向 `http://www.gamelan.com:80/pages/Gamelan.network.html`。其中 80 代表主机上的端口号。

如果参数不正确,构造方法会抛出 `MalformedURLException` 异常。所以,在创建 `URL` 对象时应使用 `try...catch` 结构。`URL` 对象创建成功后将不能被改变。

**例 12.13** 查看 `URL` 的属性,如图 12.13 所示。

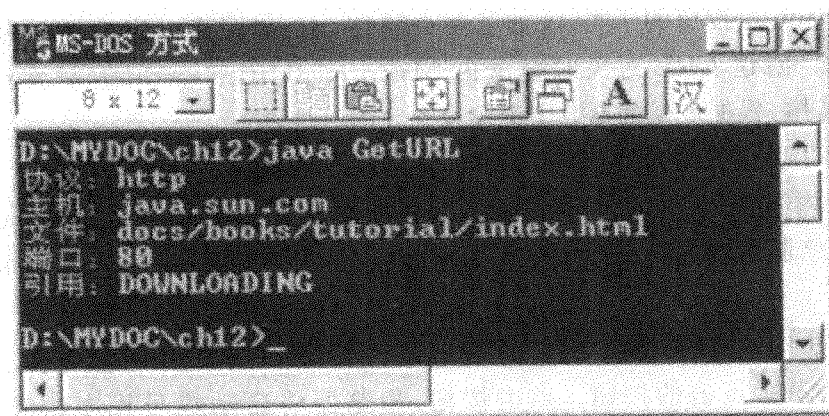


图 12.13

```
import java.io. * ;
import java.net. * ;

public class GetURL {
    public static void main(String[] args) throws Exception {
        URL url = new URL("http","java.sun.com", 80,
            "docs/books/tutorial/index.html# DOWNLOADING");
        System.out.println("协议:" + url.getProtocol());
        System.out.println("主机:" + url.getHost());
        System.out.println("文件:" + url.getFile());
        System.out.println("端口:" + url.getPort());
        System.out.println("引用:" + url.getRef());
    }
}
```

### 12.3.4 利用 `URL` 读取服务器文件

创建 `URL` 对象后,可调用 `openStream` 方法从 `URL` 读取输入流,如下面的例子。

**例 12.14** 读取服务器文件,结果如图 12.14。

```
import java.io. * ;
import java.net. * ;
```

```

public class URLReader {
    public static void main(String[] args) throws Exception {
        URL url=new URL("http://www.zz.ha.cn/");
        InputStreamReader urlStream=new InputStreamReader(url.openStream());
        BufferedReader in=new BufferedReader(urlStream);
        String str;
        while ((str=in.readLine())!=null)
            System.out.println(str);
        in.close();
        urlStream.close();
    }
}

```

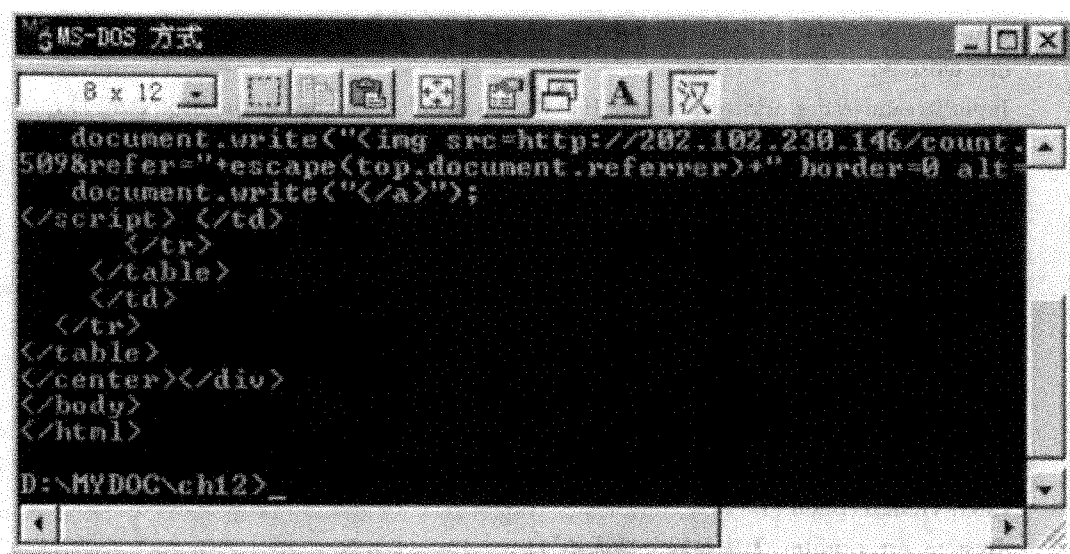


图 12.14

运行这个程序时,将启动拨号程序连接到主机 <http://www.zz.ha.cn>。然后,你就看到首页的文本内容一行一行出现在屏幕上。这里使用了缓冲读取器,URL 输入流作为参数而被缓冲,可以调用缓冲读取器的 `readLine` 方法每次读取一行并显示出来。从这个例子中你会发现访问网络资源原来如此简单。

### 12.3.5 利用 `URLConnection` 和服务端交互

如果希望既能接收又能发送信息,可使用 `URLConnection` 类。`URL` 的 `openConnection` 方法可以返回一个 `URLConnection` 对象,这个对象在客户机和服务器之间建立了一条数据通道,可进行双向数据传输。

**例 12.15** 向服务器发送信息并接收信息处理结果,如图 12.15 所示。

```

import java.io.*;
import java.net.*;

```

```

public class Reverse {
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("用法: java Reverse \"这里是一个字符串\"");
            System.exit(1);
        }
        String sendMessage = URLEncoder.encode(args[0]);

        try {
            URL url = new URL("http://java.sun.com/cgi-bin/backwards");
            URLConnection channel = url.openConnection();
            channel.setDoOutput(true);
            PrintWriter out = new PrintWriter(channel.getOutputStream());
            out.println("string=" + sendMessage);
            out.close();
            InputStreamReader urlStream = new InputStreamReader(channel.getInputStream());
            BufferedReader in = new BufferedReader(urlStream);
            String getMessage;
            while ((getMessage = in.readLine()) != null)
                System.out.println(getMessage);
            in.close();
            urlStream.close();
        } catch (MalformedURLException e) {
            System.err.println("无法建立 URL");
            System.exit(1);
        } catch (IOException e) {
            System.err.println("无法和服务端通信");
            System.exit(1);
        }
    }
}

```

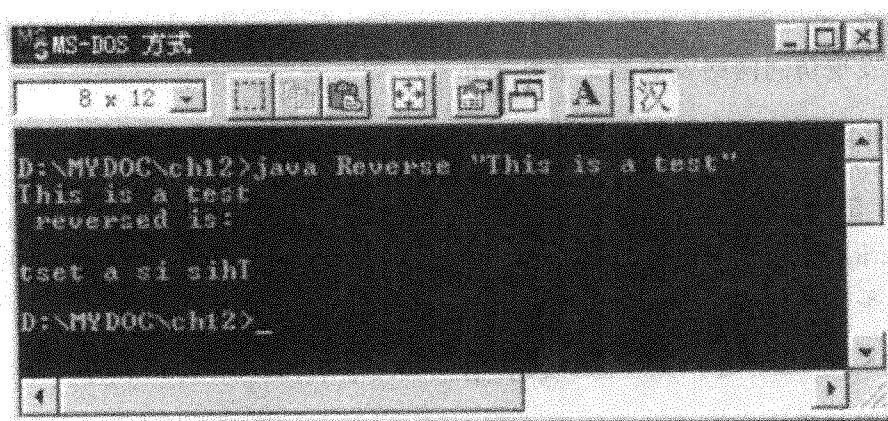


图 12.15

Sun 公司的网络服务器上有一个小程序,可将客户机发来的字符串进行翻转处理。程序进行了测试,发送的字符串是“This is a test”,翻转结果是“tset a si sihT”。

程序首先处理命令行参数。如果命令行参数长度不为 1,说明用户没有输入参数或输入的参数太多,就退出程序并显示用法。如果正确输入了一个命令行字符串,就对它进行编码处理。程序调用了 URLEncoder 的 encode 方法,将代表字符串的参数 args[0] 进行编码,去掉字符串中的非字母内容,这是服务器的要求。

然后,创建一个 URL 对象 url 并指向 http://java.sun.com/cgi-bin/backwards,再调用 url 的 openConnection 方法创建一个 URLConnection 对象 channel,这样就建立了客户机和服务器的连接。

接着,创建打印写入器对象 out,输出流由 channel 的 getOutputStream 方法创建。如果服务器支持客户机的输出流,就会在服务器端产生标准输入流。程序调用打印流的 println 方法发送数据,然后关闭输出流。这里,你会看到向服务器发送数据就像在本地计算机屏幕输出一样简单。从服务器读取信息的方法和例 12.14 一样。

从这个例子中我们归纳出向服务器发送信息的步骤:

- (1) 创建 URL 对象;
- (2) 建立连接到 URL 对象的通道;
- (3) 指定这个通道可输出;
- (4) 指定通道所用的输出流,输出流将和服务器的标准输入流连接;
- (5) 向输出流写入数据;
- (6) 关闭输出流。

### 12.3.6 利用 Socket 和服务交互

URL 可建立高层即应用层连接,而 Socket 可用来建立低层连接,它们都是基于 TCP 协议的。连接双向数据通道的端点称为 Socket,常用于编写客户机/服务器应用程序。Socket 要和一个端口号捆绑在一起,所以又称为套接字,因为端口号是用一个数字来表示的。

这种双向通信要在服务器端和客户机端分别编程并分别运行。服务器端首先要建立一个 ServerSocket,以指定端口号并监听客户机的请求,还要建立一个 Socket 用来和客户机通信。客户机端则要建立套接到同一个端口的 Socket 以便和服务器通信。

例 12.16 两台计算机进行聊天,下面是服务器端程序。图 12.16 是服务器收发信息的情况,图 12.17 是客户机收发信息的情况。

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class Server extends Frame implements ActionListener {
    Label label=new Label("交谈");
```

```

Panel panel=new Panel();
TextField tf=new TextField(10);
TextArea ta=new TextArea();
ServerSocket server;
Socket client;
InputStream in;
OutputStream out;

public Server() {
    super("服务器");
    setSize(250,250);
    panel.add(label);
    panel.add(tf);
    tf.addActionListener(this);
    add("North",panel);
    add("Center",ta);
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    show();

    try {
        server=new ServerSocket(4000);
        client=server.accept();
        ta.append("已连接的客户机:"+client.getInetAddress().getHostName()+"\n\n");
        in=client.getInputStream();
        out=client.getOutputStream();
    } catch (IOException ioe) {}

    while(true) {
        try {
            byte[] buf=new byte[256];
            in.read(buf);
            String str=new String(buf);
            ta.append("客户机说:"+str);
            ta.append("\n");
        } catch (IOException e) {}
    }

    public void actionPerformed(ActionEvent e) {
        try {

```

```

        String str=tf.getText();
        byte[] buf=str.getBytes();
        tf.setText(null);
        out.write(buf);
        ta.append("我说:"+str);
        ta.append("\n");
    } catch (IOException ioe) {}
}

public static void main(String[] args) {
    new Server();
}
}

```

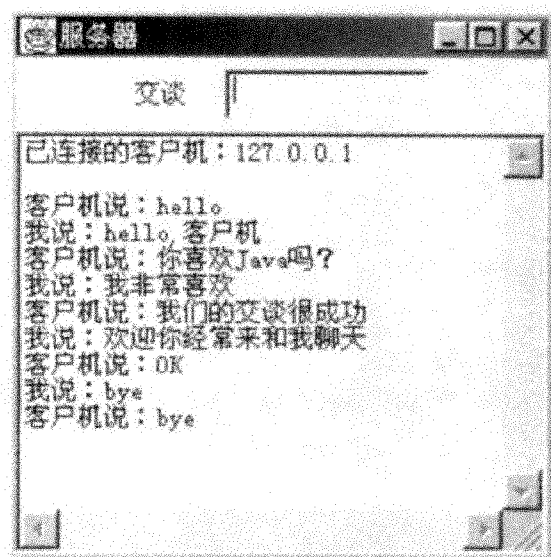


图 12.16

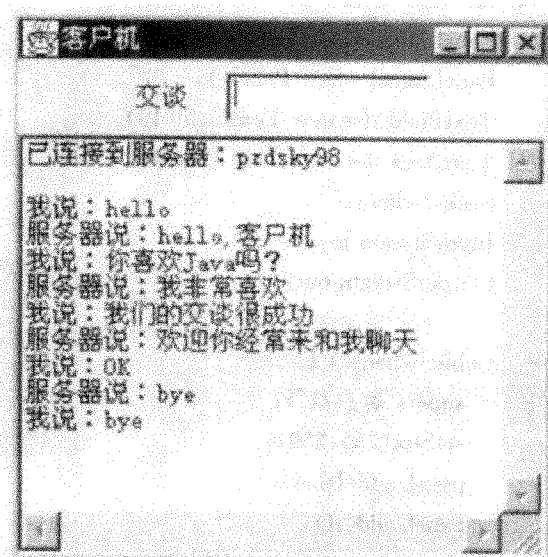


图 12.17

本程序是一个窗口应用程序,在窗口上添加了一个文本框和一个文本区,文本框用来输入信息,文本区用来显示接收和发送的信息。

在构造方法的开始部分设定了窗口的基本属性,添加各种对象,并将自身注册为文本框的事件监听者。当用户在文本框里按下回车键时将激发键盘动作事件,程序定义了该事件处理方法。使用匿名类为窗口添加关闭功能。

构造方法的第二部分是一个 try...catch 块,包含了服务器端编程的关键内容。创建了一个 ServerSocket 对象 server 并指定端口 4000,然后建立 Socket 对象 client,这个对象不能用构造方法创建,是由 server 的 accept 方法创建的。创建过程是这样的:服务器程序运行后,将处于阻塞状态,一直等到一个客户机连接到这个端口,服务器端的 client 就和这个客户机联系起来了,也就是说双向通道此时才建成。

client 的 getInputStream 和 getOutputStream 方法可以创建输入流和输出流,我们将输入流 in 和输出流 out 分别与它们对接。

构造方法的最后部分是一个条件恒为真的 while 循环,其中包含一个 try...catch 块,其作用是重复接收客户机发来的信息,将接收的字节流转换成字符串显示出来。

在键盘动作事件处理方法中,主要是把用户在文本框输入的内容发送给客户机。文本框的内容先转换成字节数组,然后将这个字节数组写入输出流,发送给客户机。最后将已发送的内容显示在文本区中。

**例 12.17** 客户机端程序,和服务器聊天,参见图 12.17。

```
import java.io. * ;
import java.net. * ;
import java.awt. * ;
import java.awt.event. * ;

public class Client extends Frame implements ActionListener {
    Label label=new Label("交谈");
    Panel panel=new Panel();
    TextField tf=new TextField(10);
    TextArea ta=new TextArea();
    Socket client;
    InputStream in;
    OutputStream out;

    public Client() {
        super("客户机");
        setSize(250,250);
        panel.add(label);
        panel.add(tf);
        tf.addActionListener(this);
        add("North",panel);
        add("Center",ta);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e){
                System.exit(0);
            }
        });
        show();

        try {
            client=new Socket(InetAddress.getLocalHost(),4000);
            ta.append("已连接到服务器:"+client.getInetAddress().getHostName()+"\n\n");
            in=client.getInputStream();
            out=client.getOutputStream();
        } catch (IOException ioe) {}
    }
}
```



```

while(true) {
    try {
        byte[] buf=new byte[256];
        in.read(buf);
        String str=new String(buf);
        ta.append("服务器说："+str);
        ta.append("\n");
    } catch (IOException e) {}
}

public void actionPerformed(ActionEvent e) {
    try {
        String str=tf.getText();
        byte[] buf=str.getBytes();
        tf.setText(null);
        out.write(buf);
        ta.append("我说："+str);
        ta.append("\n");
    }
    catch (IOException ioe) {}
}

public static void main(String[] args) {
    new Client();
}

```

客户机程序和服务器程序在窗口方面的设计完全一样,信息的接收和发送处理也基本相同,差别主要在于通信连接方面。

客户机程序创建了 Socket 对象 client,是用构造方法创建的。构造方法的第一个参数应该是一个服务器地址,因为这两个程序要在同一台机器上运行,所以直接用 InetAddress 的 getLocalHost 方法获取本机地址。第二个参数是端口号 4000,这里要和服务器指定的端口号保持一致才能连接到该端口。如果在网络上运行客户机程序,可将地址改为网络服务器地址或名称,这个服务器必须运行上述服务器端程序。

注意:首先运行服务器程序,然后运行客户机程序,这要打开两个 MS-DOS 窗口才能实现同步运行。如果先运行客户机程序或服务器没有准备好,Java 虚拟机将会抛出找不到服务器异常。

如何指定端口号呢? 这里服务器起主要作用,客户机只要和服务器保持一致即可。指定端口号时要遵循一个原则,不要使用 1024 以内的端口,它们是被系统保留的,如下:

- 07 号端口为默认的服务端口,提供常用服务;
- 80 号是 WWW 服务端口;

- 25 号是电子邮件服务端口；
- 21 号是 ftp 端口；
- 513 号是远程登录端口。

### 12.3.7 利用 Datagram 和服务交互

Socket 连接一经建立,在未关闭前一直占用系统资源,不管有没有信息收发。这种情形就像打电话,拨通以后双方可以一直通话,也可以不说一句话。

有很多应用并不需要维持一个固定连接,如收发电子邮件就不需要在两台计算机上建立固定信息通道,发信人一旦发出邮件就可断开网络连接,收信人则可在任意时间接收邮件。这种情况下,使用基于无连接的数据报 Datagram 就比较合适,而且更加简单。数据报是独立的、内含地址信息的数据包,它的到达、到达时间和内容都没有保证,但资源占用较少。

Java 有两个数据报类:DatagramSocket 和 DatagramPacket,程序可通过 DatagramSocket 收发 DatagramPacket,DatagramPacket 也可以利用广播的方式向多台计算机发送。我们看下面利用数据报进行交互的例子。

例 12.18 服务器端程序,接收客户机信息也向客户机发送信息。图 12.18 是服务器收发信息的情况,图 12.19 是客户机收发信息的情况。

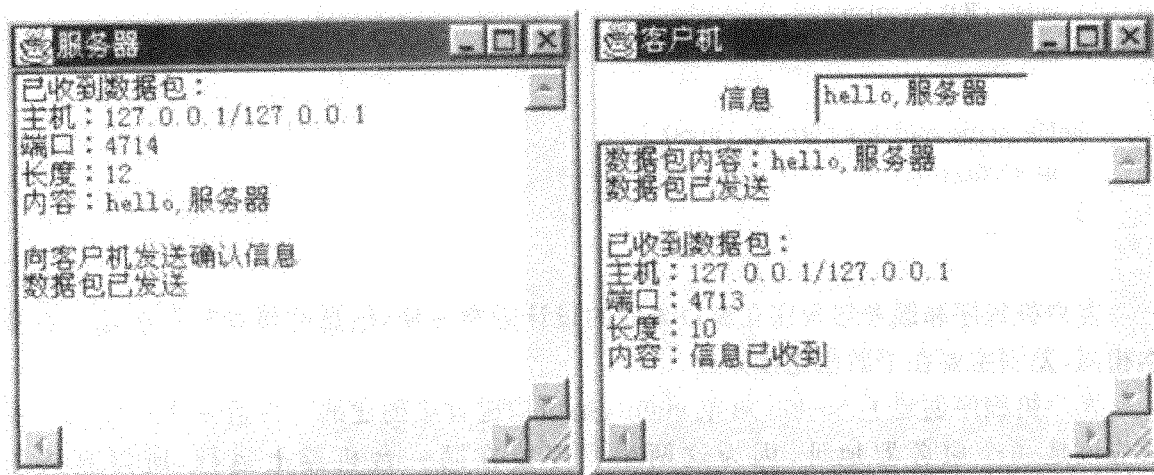


图 12.18

图 12.19

```
import java.io.*;
import java.net.*;
import java.awt.*;
import java.awt.event.*;

public class DataServer extends Frame {
    TextArea ta=new TextArea();
    DatagramSocket in,out;
    DatagramPacket dataIn,dataOut;
```

```

public DataServer() {
    super("服务器");
    setSize(250,200);
    add("Center",ta);
    addWindowListener(new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
    show();

    try {
        out=new DatagramSocket();
        in=new DatagramSocket(4000);
        byte data[]=new byte[256];
        dataIn=new DatagramPacket(data,data.length);
        in.receive(dataIn);

        String str=new String(dataIn.getData());
        ta.append("已收到数据包:\n");
        ta.append("主机:" + dataIn.getAddress() + "\n");
        ta.append("端口:" + dataIn.getPort() + "\n");
        ta.append("长度:" + dataIn.getLength() + "\n");
        ta.append("内容:" + str);

        str="信息已收到";
        byte b[]=str.getBytes();
        dataOut=new DatagramPacket(b,b.length,dataIn.getAddress(),5000);
        out.send(dataOut);
        ta.append("\n\n向客户机发送确认信息");
        ta.append("\n数据包已发送");
    } catch (IOException ioe) {}
}

public static void main(String[] args) {
    new DataServer();
}

```

本程序是一个窗口应用程序,在窗口上添加了一个文本区,用来显示接收和发送的信息。在构造方法的开始部分设定了窗口的基本属性,添加文本区对象,使用匿名类添加关闭窗口功能。

构造方法剩下的部分是一个 try...catch 块,包含了全部信息收发功能。输出口 out

在创建时没有指定端口号,这样它就可以使用机器上任何可用端口。输入口在创建时指定端口号为 4000。

定义一个字节数组,长度为 256 个字节。创建接收数据包对象 dataIn 时,使用字节数组为数据包的缓冲区。然后调用输入口 in 的 receive 方法监听客户机的连接,输入口 in 只监听 4000 号端口,一旦有数据传递过来,就把它们存入 dataIn 的缓冲区里。

数据包有很多方法;getData 可取出数据,etAddress 返回客户机的地址、getPort 返回客户机发送数据使用的端口号,.getLength 返回数据长度。程序取出了这些参数并显示在文本区中。

最后,向客户机发送确认信息,表明数据已经收到。创建发送数据包 dataOut 时,使用了已经准备好的字节数组,包含的信息是转换以后的字符串“信息已收到”。构造方法中有 4 个参数:字节数组、字节数组的长度、客户机地址(这里用 dataIn 包含的地址)、客户机上指定的接收端口号 5000。dataOut 创建以后,调用输出口 out 的 send 方法发送出去。

**例 12.19** 客户机端程序,接收服务器信息也向服务器发送信息。参见图 12.19。

```
import java.io. * ;
import java.net. * ;
import java.awt. * ;
import java.awt.event. * ;

public class DataClient extends Frame implements ActionListener {
    Label label=new Label("信息");
    Panel panel=new Panel();
    TextField tf=new TextField(10);
    TextArea ta=new TextArea();
    DatagramSocket in, out;
    DatagramPacket dataIn,dataOut;

    public DataClient() {
        super("客户机");
        setSize(250,200);
        panel.add(label);
        panel.add(tf);
        tf.addActionListener(this);
        add("North",panel);
        add("Center",ta);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
        show();
    }
}
```

```

try {
    out=new DatagramSocket();
    in=new DatagramSocket(5000);
    byte data[]=new byte[256];
    dataIn=new DatagramPacket(data,data.length);
    in.receive(dataIn);

    String str=new String(dataIn.getData());
    ta.append("已收到数据包:\n");
    ta.append("主机:"+dataIn.getAddress()+"\n");
    ta.append("端口:"+dataIn.getPort()+"\n");
    ta.append("长度:"+dataIn.getLength()+"\n");
    ta.append("内容:"+str);
} catch (IOException ioe) {}

}

public void actionPerformed(ActionEvent e) {
    try {
        String str=tf.getText();
        byte b[]=str.getBytes();
        dataOut=new DatagramPacket(b,b.length,InetAddress.getLocalHost(),4000);
        out.send(dataOut);
        ta.append("数据包内容:"+str+"\n");
        ta.append("数据包已发送\n\n");
    } catch (IOException ioe) {}
}

public static void main(String[] args) {
    new DataClient();
}

```

客户机程序和服务器程序基本相同,差别有以下几个方面:客户机添加了文本框以输入要发送的信息;在键盘动作事件处理方法中发送信息,使用本地主机地址,并发送到服务器指定端口 4000;客户机指定的接收口为 5000。

注意:这两个程序的运行次序无限制,但也要在两个 MS-DOS 窗口分别运行。运行以后各自处于监听状态,客户机可以主动发出信息,服务器收到信息后会自动发送一个确认信息。从图中可以看到,客户机发送信息时使用的是 4714 端口,而服务器发送信息时使用的是 4713 端口。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 Applet,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 Applet,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 Applet,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 Applet,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 `Applet`,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 `Applet`,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。



## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 Applet,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 Applet,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 Applet,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 Applet,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 `Applet`,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 `Applet`,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 `Applet`,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 `Applet`,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 `Applet`,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 `Applet`,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。

## 习 题

- 12-1 什么是 Java 的数据流?
- 12-2 编写应用程序,利用标准输入方法从键盘输入字符,并将其显示在屏幕上。
- 12-3 编写应用程序,利用缓冲输入流 `BufferedInputStream` 从键盘输入字符串,并将输入的字符串写入一个文件中。
- 12-4 编写应用程序,打开一个文本文件,并将其内容输出到屏幕上。
- 12-5 编写具有图形用户界面的应用程序,将文本区输入的内容写到一个已经存在的文本文件的开头。
- 12-6 Java 中的线程和多线程指的是什么?
- 12-7 线程有哪些状态?它们是如何转换的?
- 12-8 用继承的方法创建一个多线程程序。
- 12-9 使用接口 `Runnable` 创建一个多线程程序。
- 12-10 同步机制有什么作用?
- 12-11 编写一个龟兔赛跑的应用程序,它们各自在自己的线程中移动,用符号代表龟和兔,兔子跑得快,但有休眠,用直线表示它们跑过的路程。
- 12-12 用多线程编写一个 `Applet`,让一个红色小球从左边移动到右边,碰到边框再反弹回去。
- 12-13 如何通过 URL 从服务器读取文件?
- 12-14 什么是面向连接的网络服务?什么是面向非连接的网络服务?
- 12-15 客户机如何利用 `Socket` 与服务器建立连接?
- 12-16 客户机如何利用 `Datagram` 与服务器建立连接?
- 12-17 服务器如何通过端口监听客户机的请求?
- 12-18 编写一个 `Applet`,添加一个下拉式列表,列表的元素是 10 个 URL 地址,点击可以打开对应的网页。
- 12-19 编写程序,实现客户机和服务器的信息交互。
- 12-20 编写程序,利用多线程实现多个客户机和一个服务器的信息交互。